NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

MESSAGE PRIORITIZATION FOR ROUTING IN A DTN ENVIRONMENT

by

Christopher A. Rapin

March 2011

Thesis Co-Advisors:	Geoffrey G. Xie Robert Beverly

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2011	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Message Prioritization for Routing in a DTN Environment	**5. FUNDING NUMBERS**
6. AUTHOR(S) Christopher A. Rapin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _______N/A_________.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Networks have become an integral part of life today. However, connectivity problems arise in rural areas or battlefields where wired networks do not exist and wireless networks have limited coverage. In this regime, research into delay/disruption tolerant networking (DTN) techniques can help maintain opportunistic connectivity with eventual delivery of information. However, current generations of DTN routing techniques have several weaknesses of their own; particularly when the network is under high demand, both message losses and message delays rise quickly. This thesis investigates the potential of a message priority system to maintain delivery rate delays in proportion to message priority. Currently, the priority field exists in the standardized DTN metadata bundle header, but no implementation exists to use message priority as a forwarding criterion. In this thesis, using an eight node PC-based test-bed, we examine performance using existing DTN forwarding strategies, and then implement two new forwarding strategies of our own. Using these two new strategies we repeat the baseline testing using simulated data and disruptions, and observe the results. Our research aims to provide service estimations in terms of delivery rates and comparative delivery times for all levels of priority through all regimes of network demand.

14. SUBJECT TERMS Delay, Disruption, Tolerant, Network, Networks, DTN, DTN2, DTNRG, IETF, PRoPHET, GRTR, FIFO, COS, QOS, Class of Service, Quality of Service, Routing, Forwarding, oasys, Berkeley DB, sourceforge	**15. NUMBER OF PAGES** 183
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

MESSAGE PRIORITIZATION FOR ROUTING IN A DTN ENVIRONMENT

Christopher A. Rapin
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2011

Author: Christopher A. Rapin

Approved by: Geoffrey G. Xie
Thesis Co-Advisor

Robert Beverly
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Networks have become an integral part of life today. However, connectivity problems arise in rural areas or battlefields where wired networks do not exist and wireless networks have limited coverage. In this regime, research into delay/disruption tolerant networking (DTN) techniques can help maintain opportunistic connectivity with eventual delivery of information. However, current generations of DTN routing techniques have several weaknesses of their own; particularly when the network is under high demand, both message losses and message delays rise quickly. This thesis investigates the potential of a message priority system to maintain delivery rate delays in proportion to message priority. Currently, the priority field exists in the standardized DTN metadata bundle header, but no implementation exists to use message priority as a forwarding criterion. In this thesis, using an eight node PC-based test-bed, we examine performance using existing DTN forwarding strategies, and then implement two new forwarding strategies of our own. Using these two new strategies we repeat the baseline testing using simulated data and disruptions, and observe the results. Our research aims to provide service estimations in terms of delivery rates and comparative delivery times for all levels of priority through all regimes of network demand.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

Ack	Acknowledgement
CDF	Cumulative Distribution Function
CoS	Class of Service
DTN	Delay/Disruption Tolerant Network
DTNRG	Delay Tolerant Networking Research Group
Gb	Gigabit
GB	Gigabyte
GHz	Gigahertz
HDD	Hard Disk Drive
ID	Internet Draft
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
LAN	Local Area Network
LoS	Line of Sight
MAC	Media Access Control
MTU	Maximum Transfer Unit
PDA	Personal Digital Assistant
PRoPHET	Probabilistic Routing Protocol Using History of Encounters and Transitivity
RAM	Random Access Memory
RPM	Revolutions Per Minute
SATA	Serial Advanced Technology Attachment
SMS	Short Message Service

TCP/IP	Transfer Control Protocol/Internet Protocol
UAV	Unmanned Aerial Vehicle

ACKNOWLEDGMENTS

The author would like to acknowledge the unending patience of his wife during the entire thesis process. Her tolerance of my strange requests for help and reading pages and pages of text that made as much sense to her as the ramblings of a child made this whole process possible.

In addition, he would like to thank his thesis advisors, Professor Geoff Xie and Professor Robert Beverly, who helped a neophyte Unix user jump head first into the deep end of the pool. They managed to answer a seemingly unending stream of questions as he slowly learned to swim while not losing their patience and maintaining hope that this thesis would eventually be completed.

Further thanks go to LT Scott Huchton, USN. He acted as a sounding board for many of my ideas during this process and additionally helped a great deal when Unix command lines continually evaded my memory and Google searches.

Finally, a huge thank you to Major Chad Seagren, USMC, without whom I would probably still be trying to graph the cumulative distribution functions for the results section.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM

Military communication networks are essential to modern warfare as conducted by the United States Military. Nearly every system on the battlefield today communicates with others through various communication mediums. From Air Defense networks coordinating between multiple vehicles and missiles via cable and wireless data links, to command and control networks set up quickly to support decision making at all levels of command, these networks are vital to an effective campaign.

Unfortunately, these networks are also under constant stress. Enemies are seeking ways to disrupt the communications and command and control links, units moving rapidly find themselves out of range of the radios they carry, and often the physical realities of the battlespace can also create line of sight or shadowing issues for communications networks. How can we deliver communications reliability to all aspects of the military, from the soldier on the ground to the general in the Pentagon? Disruption tolerant networking offers promise in this respect.

Delay/Disruption Tolerant Networks (DTNs) have been studied extensively in the past decade. With the explosion of mobile computing platforms, and their extensive penetration through all segments of society, there now exists a vast untapped resource pool of computing power, connectivity, and memory. If leveraged properly, this could deliver connectivity and data to previously underserved or completely disconnected populations. In a battlefield setting, DTNs could deliver data services where standard radio frequency communications fail due to enemy disruptions of friendly networks, destruction of critical nodes, or even something as simple as lack of line of sight to the receiving station.

A disruption tolerant network is one in which the network connections between nodes in the network are disrupted at certain times. These disruptions can be at random intervals in some topologies, such as people meeting on the street, or could be predictable and highly regular in others, like satellites passing overhead. As a result, the nodes in the network will only be connected for a finite amount of time before the disruption occurs again. For this reason, the time a node spends connected to another node is called a contact interval. This interval can be continuous, as in a wired network infrastructure, or as short as a few seconds in some cases. A network partitioned into two or more parts is termed “segmented.” In this case, no path exists between segmented nodes in the network. A unique feature of a DTN is that in a DTN topology it is possible for a node never to be seen again once a connection is severed. Some examples of DTN topologies as they are currently being studied are: networks of mobile devices, interplanetary communication networks, and animal or vehicle based networks. Networks of mobile devices show the most promise in the military setting as forces move to realize the idea of “every soldier a sensor.” Vehicle based networks also show promise in delivering connectivity to underserved or unserved areas of a country or isolated military units. To accomplish this, developers use the vehicle as a mobile data store to bring data from a connected area to a disconnected area for distribution locally.

While DTN topologies have great promise, there are several difficulties that arise with attempts to implement them. One of the most significant is, current applications all inherently rely on the TCP/IP protocol stack. This requires an active end-to-end path to complete the delivery of data. In a DTN topology this path may never actually exist due to the disruptions of the connections between the nodes. Even if a suitable end-to-end path does exist, it may exist for only an extremely brief period of time and thereby be rendered useless for the transmittal of data. With an end-to-end path essentially nonexistent, or in the best case, undependable, routing in DTN topologies must be done over time to achieve eventual delivery at a destination. Using eventual delivery DTN routers add the dimension of time to their routing protocols. While an end-to-end path may not exist, the message can at least traverse a portion of the network on the way to its destination. Upon arrival at the node where the network segmentation has happened, the message will then be stored and wait. When the next contact interval for the broken link happens and communication is restored, the message will continue its journey. In this manner when an end-to-end path does not exist at a moment in time, it can effectively be created over time to allow the delivery of messages.

This eventual delivery has been studied in two different methods, one is known as the multiple copy approach [1] and the other a single copy approach [2]. The single copy approach has been researched, and has many advantages. In the single copy case only one copy of the message ever exists. When this message is delivered there are no other copies to deal with as in the multiple copy case. This includes both deleting copies of the message still present in the network, and preventing nodes that have copies from continuing to forward them. Also, because we only have one copy, no bandwidth is wasted forwarding additional copies of the message through the network to nodes that already contain a copy of the message. However, the knowledge required by the routing protocol is great. It must make decisions on how to forward this message onward, but if an incorrect decision is made the only copy of the message could go in the wrong direction. This may result in a message traveling farther from the intended destination increasing delivery time, or even preventing delivery completely in a worst case scenario. The single copy approach requires a large amount of correct knowledge to implement, and therefore is only useful in highly predictable and reliable DTN systems where connections can be depended upon to be available at a set time. In a military setting, this scenario could be seen by a special operations team that can only communicate to higher headquarters during a time window when the appropriate satellite is overhead, or a submarine that does not want to give away its position and raises a communication antenna at a set time each day to check in.

The multiple copy method can be much simpler in terms of message forwarding, but brings in a new and different set of challenges. In the simplest multiple copy system, a copy of the message is sent to every node that is encountered in addition to being retained on the current node. Such a strategy hopes that at least one of the nodes that have a copy of the message will eventually come across the destination and deliver the message. This routing strategy is known as epidemic routing because the message traverses the network in the same manner as a contagion spreads through a population. The complexities introduced by epidemic routing are those of queue size, bandwidth, and message deletion. If a node holds onto every message, eventually the queue at that node will become overloaded and some messages will need to be dropped. How does the node determine which message(s) should be dropped? Similarly, if every node transmits every message, the amount of required bandwidth may exceed the contact opportunity.

Many approaches attempt to address this queue problem. One using variations of hop count, a metric recording which neighbors a certain node has encountered recently and thus is more likely to encounter in the future, has shown promise. [3,4] Another uses the expiration time field carried by the bundle itself. When the bundle has existed for a prescribed period of time, set when it is created, it is automatically deleted by any node holding a copy. Finally, some systems rely on network control messages in the shape of acknowledgements to help control the number of messages in a queue and shape which messages are forwarded to particular nodes. [3]

Additional attempts to find a compromise between the knowledge required in the pure single-copy case and the storage issues of the multi-copy case have been researched also. One example of this is the Spray and Wait [5] technique. Using this system, a set number of copies of a message are sprayed across the network using various techniques. Once the messages have been sprayed, the system waits until one of the nodes that received a copy encounters the destination to deliver it. In [5] it was shown that this both limits contention for resources as there are a set number of copies of the message in the network at any given time, and results in far fewer transmissions than epidemic routing techniques. In addition, it does not require the knowledge needed by the single copy case because no forwarding is done after the spray phase. The messages are held by the nodes that received them until they encounter the destination. All of these techniques have seen successes in limiting the growth of queues in a network, but when a network is stressed by a large number of messages, queues still become difficult to manage and performance will suffer.

With this degradation of performance under stress in mind, can we provide quality of service guarantees to users of the system? Traditional networks can provide QoS benchmarks by giving reduced priority to one user's traffic over the network and allowing a second user's data to continue unimpeded. By doing this, service providers can make promises to the end user that they will see a certain level of service, commonly either in throughput or message latency. Taking into consideration the unpredictable nature of DTN topologies, can we make similar QoS comparisons within that environment?

Using QoS in military systems that suffer from frequent or infrequent disruptions and are therefore well suited to implementation within a DTN would allow for many advantages over traditional systems. In traditional systems messages destined for an unreachable node are dropped after a very short time. In a DTN those messages are stored until such a time as the node becomes reachable or it can be sent closer to the destination. While the time to deliver a message can never be guaranteed, sending a message with higher priority could allow the assurance that the message will be delivered in the shortest possible time. By being placed in front of lower priority messages inside the queues at each node, it is forwarded more quickly, and in theory reaches its destination faster. This would prove to be extremely useful to highly mobile or isolated units, particularly scout or spotter teams. With the DTN store and forward scheme, isolated units can store reports or images of what they see tagged with priority. Then when they return to base, encounter another unit, a vehicle passes by, or an aircraft passes overhead the most important or urgent reports are passed to the encountered node first. This ensures that the most important reports are forwarded toward the commander first if the contact period is too short to allow the transfer of all stored items. In addition, by using low power signals and forwarding reports only to other nodes nearby, the risk of detection is minimized and an effective battlefield network is obtained. When nodes are within transfer range of the main force concentration, the network performs much like a standard TCP/IP network. However, when it is disrupted, by enemy jamming for example, the true advantage of a DTN topology becomes clear. Not only can communication be maintained in a degraded manner during the disruption with wireless nodes getting close enough to overcome the noise of the jamming, but once the disruption has passed the network will rebuild itself. As nodes then reconnect with other nearby nodes, they recreate the original network. Message priority on top of DTN ensures that the most important messages are always at the front of the queues to help ensure their speedy delivery in all circumstances.

The purpose of this thesis is to (1) examine DTN operation and performance under various network conditions, (2) examine different DTN routing systems and their performance, and (3) adapt the different existing routing protocols to allow networks to prioritize messages. With prioritization we hope to see if QoS for a DTN can be realized.

Users are always looking for performance metrics to define their networks. If DTN topologies can offer some form of QoS comparisons, it will make them more palatable to users especially those in the military services. By allowing the delivery of messages as soon as possible, even when the network is disrupted frequently, DTN techniques will be equally suitable for military commanders and civilian users in remote areas. In a simple case, during an attack on a friendly unit an enemy knowing of our dependence on the electromagnetic spectrum would likely try to jam radio communications thereby cutting off the unit from other units that could reinforce them. In a typical network messages sent to this unit would be dropped, with an error returned about unreachable destinations. In a DTN however, these messages from headquarters would be stored and delivered as soon as there was a break in the enemy’s jamming. In a typical DTN node however, the messages are forwarded in a FIFO order. An order to retreat and regroup could be queued after an e-mail from a soldier’s girlfriend back in the USA. This is an unacceptable situation, so the COS system is introduced to ensure that the General’s message is pushed to the front of the queue with other high priority traffic, and the love letter is delivered when no other, more important, information is left to be sent. The determination of relative priorities between bundles is a difficult question itself. In this thesis we are interested in the implementation of a system of priorities. The policy for tagging each bundle with a specific priority is beyond the scope of this thesis and left for future work.

B. RESEARCH QUESTIONS

The research in this thesis attempts to determine the answer to following questions. The first four are the primary research questions, essentially attempting to answer if QoS analogues can be made in a DTN topology. The remainder of the questions include more specific areas investigated during the research.

1. Given a real DTN topology what are the baseline performance metrics under varying network loads?
2. At what point does message overload reduce the network performance to unacceptable levels?
3. Is it possible to accommodate prioritization in a DTN routing protocol on a DTN topology to allow QoS metrics to be determined? For this thesis we will define QoS metrics as delivery delay and delivery rate.

4. What are the appropriate comparative QoS metrics seen in our topology?

5. Does the highest QoS category perform better than lower categories, even under conditions that made performance poor in the unmodified network?

6. What new security vulnerabilities does this modified network routing introduce into DTNs?

C. THESIS ORGANIZATION

This chapter provides an overview of Delay Tolerant Networking, and how the technology can be used to enhance communications in both military and civilian worlds. Also, it gives a very high level overview of the problems encountered in some of the current implementations. In addition, it outlines the research questions to be investigated over the course of this thesis. Finally, the chapter provides a look at the topics discussed through the remaining chapters in the document.

Chapter II. The background chapter provides definitions, a general overview of current technologies, and applications in use for implementing DTNs. The main purpose of this chapter is to remove any ambiguities between similar terms, and supply the reader with a solid understanding of the current protocols. In addition, it will compare each of the differing protocols that will be used in the research, explain how they differ, and the benefits and costs of each. Finally, it will explain related work in areas surrounding the thesis research topic.

Chapter III. This chapter details the experimental design used in this thesis. It details the decisions that were made when designing the ideal test bed for the proposed routing system and the tests that will be used to evaluate the system. In addition, it describes the design of the new QoS forwarding and queuing strategies we developed for experimentation. Finally, it describes the expected results we hope to attain over the testing runs.

Chapter IV. This chapter details the actual test bed used. All the details needed to set up an identical system for further testing or additional expansion of capabilities are included. Any deviation from our experimental design in Chapter III is explained. Finally, the results of our tests on the actual test bed are reported and analyzed.

Chapter V. This chapter presents the conclusions drawn from the overall thesis effort, particularly from the experimental results reported in Chapter IV. In addition it includes possible avenues for future research on this DTN routing, and areas where further research is needed to refine the results of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter illuminates the differences between connected networks today and a disruption/delay tolerant network. We begin with a brief discussion of the Internet today (as an exemplar network) and the TCP/IP stack that dominates its function. Second, we will look at disruption/delay tolerant networking (DTN), its advantages, how it differs from the traditional Internet, and how it impacts message routing. Third, we will look at three routing protocols currently in use for experimentation, and examine their individual advantages and disadvantages. Finally, we will briefly look at the ByteWalla [6] data mule system developed by The Royal Institute of Technology in Stockholm, Sweden, and the DTN2 reference architecture provided by the Delay Tolerant Networking Research Group [7] (DTNRG) within the Internet Research Tack Force (IRTF).

A. THE INTERNET TODAY

1. Today’s Internet

The modern Internet links hundreds of millions of communication devices across the planet. It has done this by using a set of protocols called the TCP/IP protocol suite. Very nearly every device that communicates over the Internet uses the TCP/IP suite to communicate, move the message through the network, and to ensure the delivery of the message at its ultimate destination. The Internet today still relies primarily on wired links; however, to a greater and greater extent wireless links are appearing, particularly at the edge. All these connections to every device, whether wired or wireless have several commonalities: they are all continuously connected or very nearly so, low delay, and low error rate links. In addition, they are relatively symmetric in their data rates both upstream and downstream. With the number of wireless providers growing, and nationwide networks expanding quickly, even wireless devices such as smart phones and PDAs have nearly continuous connections to the backbone of the Internet.

2. Packet Switching

The Internet uses a concept known as packet switching to transmit data over the network. Packets are pieces of a larger block of data that have been broken into smaller parts for transmission and given individual routing headers. These packets travel across the network independent from each other potentially being lost, or arriving at their destination out of order. The packets move across links in the network connected by routers. The packets themselves contain the data being transmitted (payload) and a header (control). The header contains the source address, destination address, and various other fields containing information about the packet and its contents. Using the destination address in the header, the routers “switch” the packet from link to link until it arrives at its ultimate destination. Packets can arrive out of order, but the end-host destination, using the header data, can reassemble them in the correct order.

The Internet as we know it today requires several assumptions in order to transmit data successfully. The first, and probably most important, is that a continuous bidirectional end-to-end path exists. A second assumption is relatively short round trips, or put another way, low delays in the network. Third, there must be symmetric data rates. This does not mean that both directions must have exactly the same data rate; however, the rates should be consistent, non-fluctuating, and be relatively close to each other. Finally, the Internet today assumes low error rates. This implies that there is little or no loss of data or corruption of data as it is transmitted through the network.

3. Protocol Layers

Messages in the Internet move through protocol layers implemented at the source and destination in addition to every router between the two. The different protocol layers in the TCP/IP Protocol stack are:

- Application layer – this layer generates data at the source node, or consumes data at the destination.
- Transport layer – this layer provides end-to-end segmentation of the data into packets. In addition, it is responsible for the reassembly of the packets into their original data blocks before being handed off to the application layer. In addition, it can also provide reliability and congestion control within the network. Finally, it provides error control and flow control of the data stream.
- Network layer – this layer provides the routing information needed to provide source to

destination routing of the packets through the intermediate routers. It also provides for fragmentation and reassembly of the packets, if necessary. Fragmentation becomes necessary when a message or file is too large to be sent in one data packet across a particular link in the path. For example, Ethernet networks frequently have a MTU of 1500 bytes. Not much of use can be delivered in this small of a package so the file must be broken into 1500 byte chunks and reassembled at the destination.

- Link layer – this layer provides link to link transmission and reception of the message pieces. In addition, it provides Media Access Control (MAC). Having multiple computers on a network would be difficult if they all started transmitting at the same time. MAC at the link layer allows computers to coordinate when and who will transmit so interference is minimized. Some examples of link layer protocols include Ethernet, Point-to-Point Protocol, Token ring, and IEEE 802.11 wireless LAN.
- Physical layer – this layer provides the link to link transmission and reception of the bit streams. It also includes the modulation scheme used to send the information over whatever medium the two communicating nodes are capable of using. Some examples of physical layer mediums currently in use are copper cable, optical fiber, coaxial cable, and the radio frequency spectrum.

In Figure 1, we show a visual example of the different layers in the TCP/IP protocol suite. The source and destination nodes implement all five layers, while the routers between them only use the bottom three during routing operations. Note that the physical layer is different between each hop in the chain. A continuous path over one type of physical medium is not required. In addition, the link layer between the final router and the destination are different from the previous two hops. This could be the result of the first physical layer being Ethernet cables, the second being DSL signals sent over phone lines, and the third being wireless signals sent to a mobile endpoint. As long as all nodes are running the TCP/IP protocols, and the underlying assumptions listed previously are maintained, transmission of data will be possible.

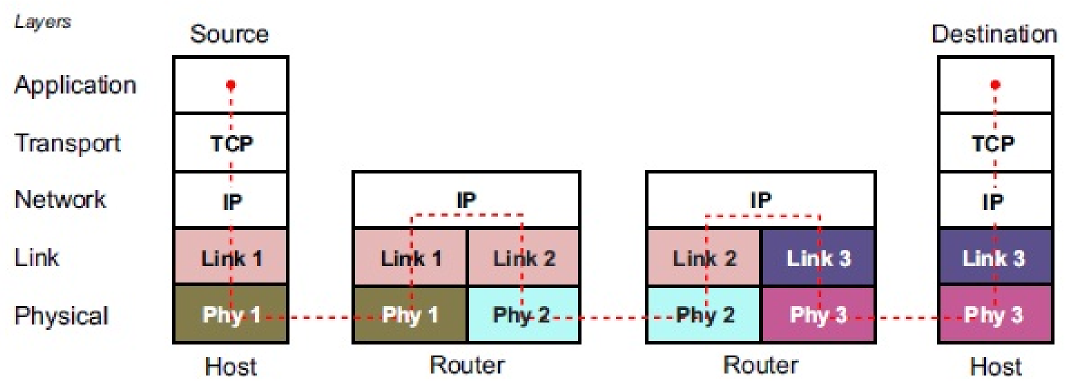

Figure 1: An Example of TCP/IP Protocol Layers [From 8]

4. Encapsulation

As data is transmitted across the Internet it is sent as a series of packets. The idea that these packets are a data payload and a header is overly simplistic. In actuality, each packet is a payload of data from the application layer which has a header placed on it by the transport layer. At the network layer, the application data and the transport header, now known as a TCP segment, are treated as the data payload, and a network header is added. This data item proceeds through the rest of the layers being further encapsulated at each one. When the packet arrives at the destination and/or intermediate router, the headers are stripped off and the resulting data is passed up the stack.

For instance, at a router, the physical layer receives the bit stream on a cable and once it has completed a packet, passes it to the link layer. The link layer then strips off the header and passes the resulting the data to the network layer. The router examines the network header to determine the destination, and then decides how to forward the packer. Once this is done, the link layer passes the packet back to the link and physical layer for re-encapsulation and forwarding. This hierarchy of encapsulation is present in all packets sent using the TCP/IP protocol. Figure 2 shows an example of this as application data is broken up and layer specific headers applied as it transits the TCP/IP stack down to the link layer. Once the link layer is reached, the data is sent as a bit stream on the physical medium available to the system.

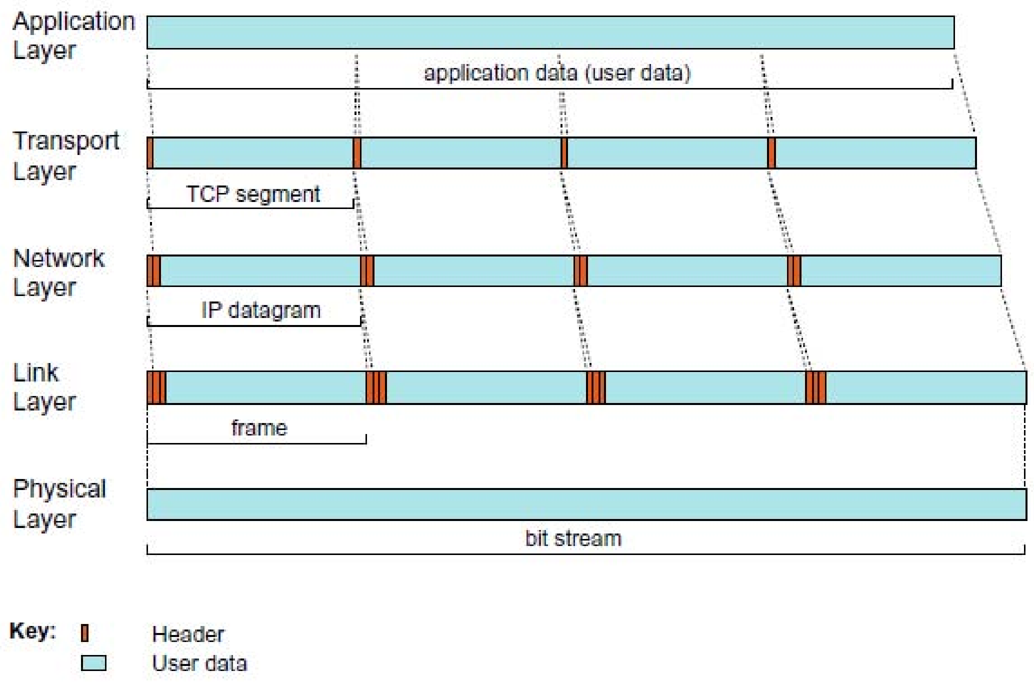

Figure 2: Visualization of Data Encapsulation [From 8]

5. Conversational Protocols

The TCP protocol is considered to be a conversational, or connection oriented, protocol because a complete transmission session involves multiple signaling round trips. These signals can be broadly grouped into three types. The setup or hello phase, the data transfer phase, and the goodbye or tear-down phase. The setup phase consists of a three-way handshake between the sender and receiver. The data transfer phase consists of the sender transmitting TCP segments and the receiver acknowledging each one. Finally, the tear down phase consists of a four way handshake that is completed before the connection between the sender and receiver is terminated. In Figure 3 the messages between the sender and receiver at the TCP layer can be seen in addition to the entire TCP/IP stack. It becomes obvious that several messages are needed to set up and tear down of the virtual circuit before and after data is sent. In addition, an acknowledgement is required for every packet sent. If the conversation is ever broken by delays, or one host disconnecting from the network, packets are lost, and communication discontinues.

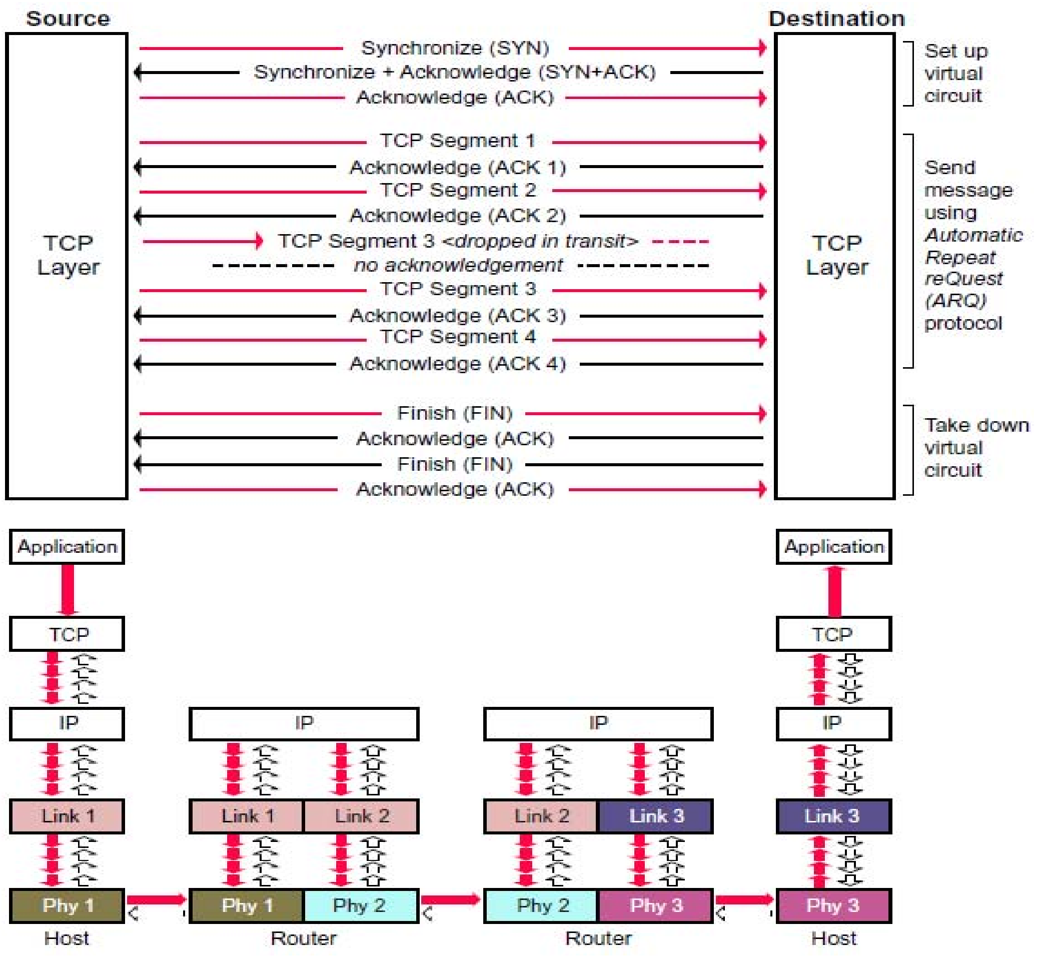

Figure 3: Conversational Protocol [From 8]

B. DELAY TOLERANT NETWORKS

1. Why a Delay/Disruption Tolerant Network?

With the evolution of wireless communications, different kinds of networks are beginning to be realized outside of the connection-oriented Internet as we know it today. These networks are developing as separate and specialized networks. Inside these specialized networks, they are relatively homogeneous, and often will have unique communications requirements that the modern Internet cannot support due to limitations in the TCP/IP protocol suite under extreme conditions. Some examples of emerging networks are:

1. Sensor networks – In these networks, communication is limited between nodes, often to prolong limited battery life by both reducing signal strength, limiting operational time, or both. In addition, if the nodes are mobile, there can be long periods of disconnection where a node is unable to communicate with neighbors due to distance.
2. Satellite Networks – These networks are characterized by large delays due to distance between the nodes. Also, error rates begin to rise due to interference from solar phenomena and limited signal strength.
3. Military Networks – connecting ships, planes, troops, and sensors, often with highly varying delay, connectivity, or communication requirements.

Connecting two of these disparate networks, or simply connecting one of them to the Internet, requires the use of an intermediary or gateway to translate between the differing communication protocols and act as a buffer for messages when the delays on either side of the gateway are significantly mismatched.

2. Advantages of Delay Tolerant Networks

The emerging network types mentioned in the previous section, and many more that are potentially evolving, do not conform to the assumptions necessary for the implementation of the TCP/IP protocol stack in use today. These networks have the challenges of being connected intermittently, having long or widely variable delays, large asymmetries in the data rates for each transmission direction, and high error rates.

When an end-to-end path from a source to a destination does not exist, the network is partitioned. This means that for some reason (link failure, Line of Sight issues, signal strength, etc.), nodes in one portion of the network cannot connect to nodes in another. Since the TCP/IP protocol suite requires an end-to-end path to function correctly, it is unsuitable here, and other protocols must be used.

Long delays for packets being sent on the network, or long variable queuing delays, can also defeat the TCP/IP suite. It can also cause problems for applications that require the quick responses that are typically seen in the Internet today, thanks to the TCP/IP protocols.

The Internet as we know it today supports moderate asymmetries. For example, commercial Internet providers force asymmetries on their customers, giving them a larger segment of bandwidth to download data, and a significantly smaller one to upload data to the Internet. However, the extremely large asymmetries that are present in some emerging networks defeat the TCP/IP suite in the same way that the large/variable delays do.

Finally, large error rates require that the errors be corrected. If the errors are to be fixed at the destination, this requires more processing at the source and extra information included with the data to allow the destination to correct any errors. This is extra overhead, and reduces the amount of data that can be sent in each packet. Another way of doing error correction is just to resend the entire packet. This does not require the same increase in overhead on a packet by packet basis, but does increase network traffic more so than the first method. Both of these options are suitable if the links have the low error rates we assume in the Internet; however, they break down and come to a halt as error rates on the paths increase. Given a link error rate, as the number of hops a packet has to make increases the error rate increases. This results in more retransmissions as the packet must successfully navigate every link in the path error free as in the Internet. In addition, this example only considers the delivery of a packet to the destination. For delivery to be considered successful, an acknowledgement of delivery must be received at the source. If this Ack packet lost or corrupted, the original data must be sent again despite being successfully received already. If a packet is delivered hop-by-hop, as in a DTN, rather than end-to-end, then when an error occurs, the packet need only be resent over the last hop instead of the whole path. This results in a linear increase in retransmissions in a hop-by-hop delivery scheme vice the exponential increase in end-to-end schemes as the hop count grows.

3. Delay Tolerant Networking

Delay tolerant networking provides us an overlay that can span any of the above challenged networks and/or the Internet as we know it today. It allows the transmission of data using a store-and-forward scheme rather than end-to-end transmission. In some cases, especially when referring to mobile nodes, the store-and-forward scheme is referred to as store-and-carry-forward. The basic ideas and concepts are the same in both cases, but store-and-carry-forward more clearly includes the concepts of mobile vice fixed nodes in the network. Figure 4 shows an example of a store-and-forward scheme. In such a scheme, every node has some kind of persistent storage provided for the storage of data that cannot be immediately sent. When data is ready to be sent, it is forwarded across the network hop by hop. If the next hop in the path is unavailable or busy, the packet is stored at the current node until resources for transmission become available. The packet is forwarded in this way until it eventually reaches its destination. The storage at each node differs from the storage used in the Internet however. In a store-and-forward scheme, the data storage is a hard disk or some other kind of persistent storage rather than the memory chips commonly found in routers. DTN nodes need persistent storage for many reasons including: the communication link to the next node may be unavailable for an extended period of time, one node in the communicating pair may send or receive data much slower than the other, or due to high error rates the message may need to be retransmitted multiple times before it is successfully received at the next node or the destination.

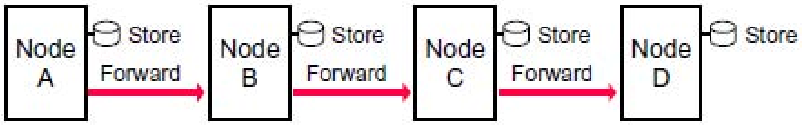

Figure 4: Visualization of Store and Forward Delivery [From 8]

Communication devices are increasingly becoming mobile, and when nodes are in motion, there is always the chance that communication could be obstructed by a foreign body such as a tree, building, or even the curvature of the earth in the case of a satellite communicating with a ground station. The Internet protocols lose data when connectivity is interrupted. Packets that are not immediately forwarded to the next node are usually dropped as space in the routers queue is at a premium. Delay tolerant networks can support communications between nodes that are connected intermittently by using the store and forward technique above.

With connections that are unavailable at times, periods of connectivity are known as “contacts” or “encounters.” There are two general types of contacts used in DTNs. These two types are opportunistic contacts and scheduled contacts. Opportunistic contacts happen at unscheduled times and last for unpredictable intervals. When personal mobile devices come in range of each other as people on the street meet, it is an example of an opportunistic contact period. Any device that is in motion can make use of opportunistic contacts whenever they happen to be within range of another node with line of sight suitable for communication.

In Figure 5 we provide a visual example of the concept of opportunistic contacts. The two red dots represent mobile nodes. Initially, the two nodes can communicate since they are close enough. In the second diagram, one mobile node has climbed a hill and is now out of range of the other. However, with his new position on the top of the hill he can communicate with a fixed tower.

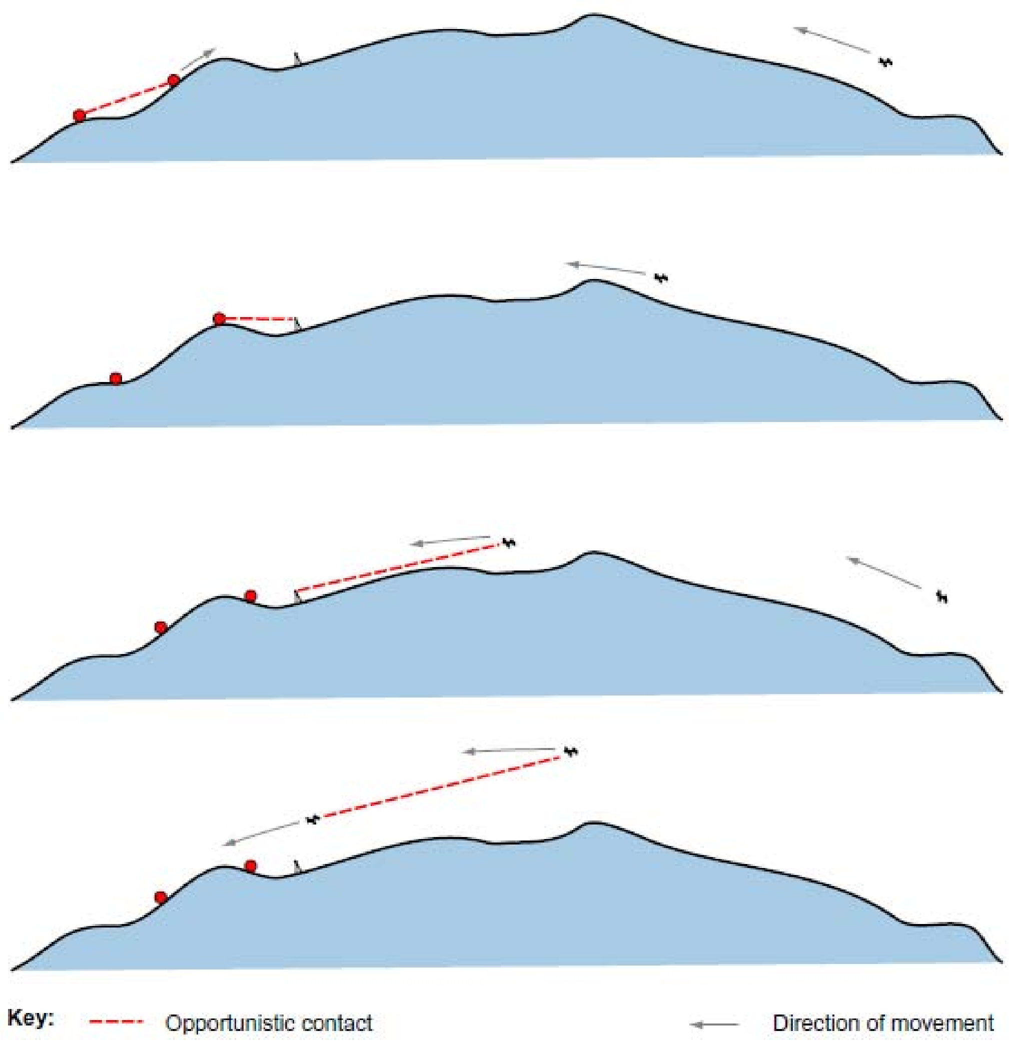

Figure 5: Concept Of Opportunistic Contacts [From 8]

In the last two frames aircraft come into the area, these can be UAVs, reconnaissance aircraft, or any aircraft equipped to communicate effectively with the ground stations. In the third frame, an aircraft comes over the horizon from the point of view of the tower and can then communicate with it. In the last frame, the two aircraft have come close enough that they can communicate between each other. This diagram is not meant to imply that only one connection can be in effect at a time, but merely to show that opportunistic contacts are by nature unpredictable.

Scheduled contacts, on the other hand, often require time synchronization through the entire network as nodes will come online only at prescheduled times. Reactive measures, where a node announces its next availability period before disconnecting, are possible also, but more often, a predetermined schedule is used. Nodes using scheduled contacts only need to try and connect during periods of time that another node is available. For example, effecting communication between planets entails extremely long delays; however, the orbits and rotations of planets and their satellites are highly predictable. A message can be sent while the receiving node is obstructed so long as the speed of light delay results in the message arriving when the receiver is unmasked.

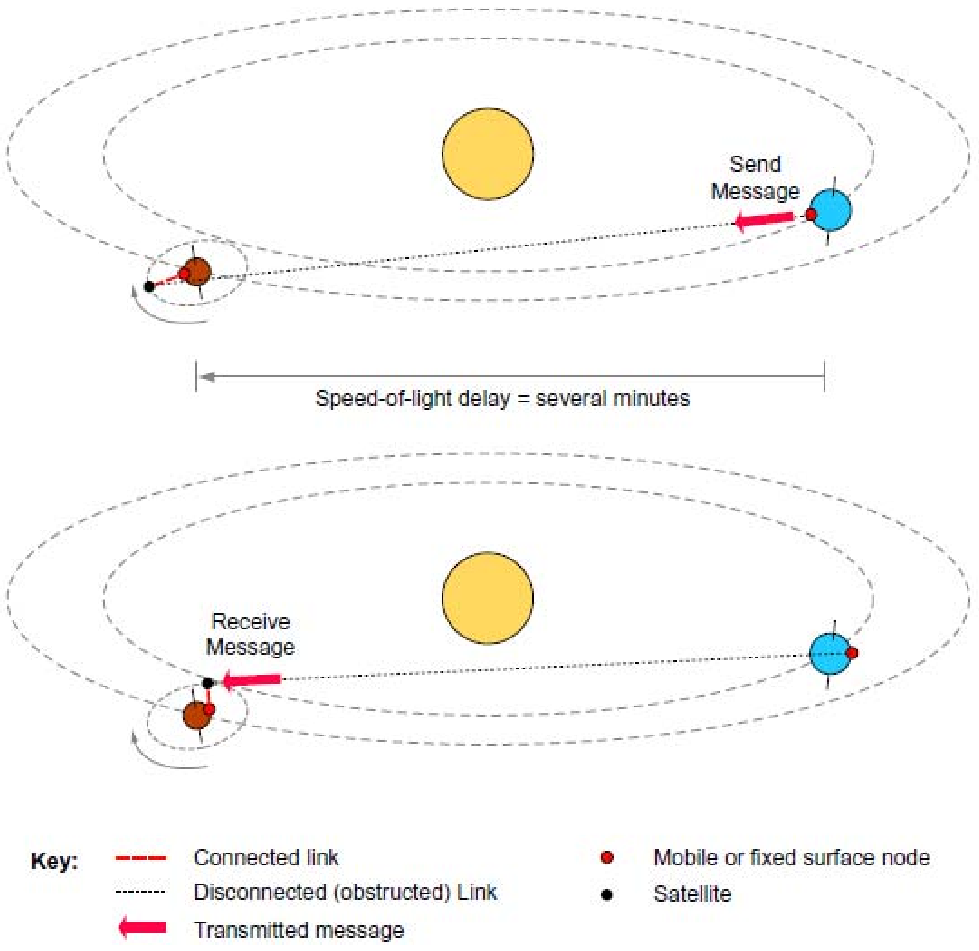
Send
Message
Speed-of-light delay = several minutes
Receive
Message
Key:
Connected link
Disconnected (obstructed) Link
Transmitted message
Mobile or fixed surface node
Satellite

Figure 6: Concept of Scheduled Contacts [From 8]

In Figure 6, we see an example of interplanetary communication where in the first frame a message leaves one planet and begins its trip to another. When the message is sent, the receiver is on the back side of the planet unable to receive the message. However, the propagation time the message takes transiting the distance between the two planets results in the receiver being unmasked when the message arrives. As the time moves on the message eventually reaches the satellite orbiting the planet and is relayed to the surface node. This type of communication is possible because the sender knows precisely when the receiver will be in a position to receive messages. The sender can begin sending information before the receiver is unmasked as long as it ensures that the receiver is unmasked when the information finally arrives.

To accomplish the store-and-forward techniques necessary to move messages hop by hop over intermittently available links, delay tolerant networks implement an additional protocol layer into the standard five layer model. This layer is the bundle layer [9], and it ties together all of the disparate networks and communication methods below it, allowing delay tolerant networks to run as an overlay over nearly any communications method.

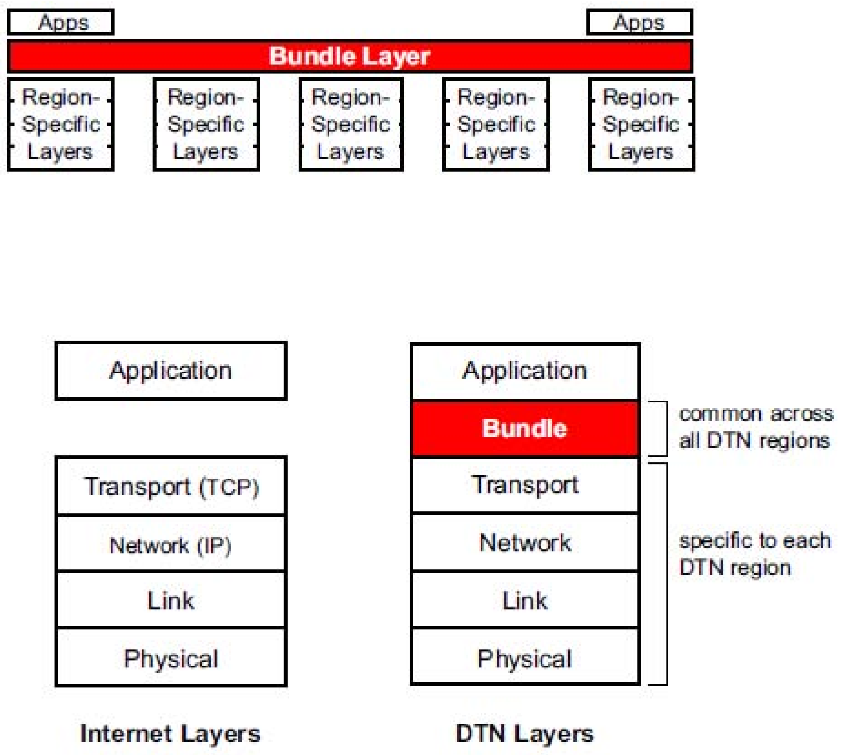

Figure 7: Visualization of the Bundle Layer [From 8]

The application layer generates the data to be sent and then passes it to the bundle layer. The bundle layer then encapsulates the data into a single bundle. This bundle is then handled by the following four layers in accordance with the underlying network technology. If it is TCP/IP, the bundle is broken into packets, encapsulated, and forwarded just as any other Internet traffic. If some other network is being supported, the transport, network, link, and physical layers would be implemented as appropriate to enable transport through this network.

Like every other layer of the protocol stack, the bundle layer encapsulates the data and attaches a header. The bundle itself consists of three items. The first is the source application’s user data. The next is control information. This is provided for the destination application by the source application and instructs it on how to handle the data. Finally, there is the bundle header that is created by the bundle layer. Unlike packets, a bundle can be arbitrarily long. As the protocol stack is extended, the encapsulation of the data is also extended by the bundle layer.

In Figure 8, we see the encapsulation of information previously discussed; however, we have now added the bundle layer into the system. It can be seen that a bundle is simply the encapsulation of all user data into one bundle with the appropriate headers for delivery by the bundle agent on the receiver. It is then broken up and encapsulated by the lower stack layers in the same manner as before.

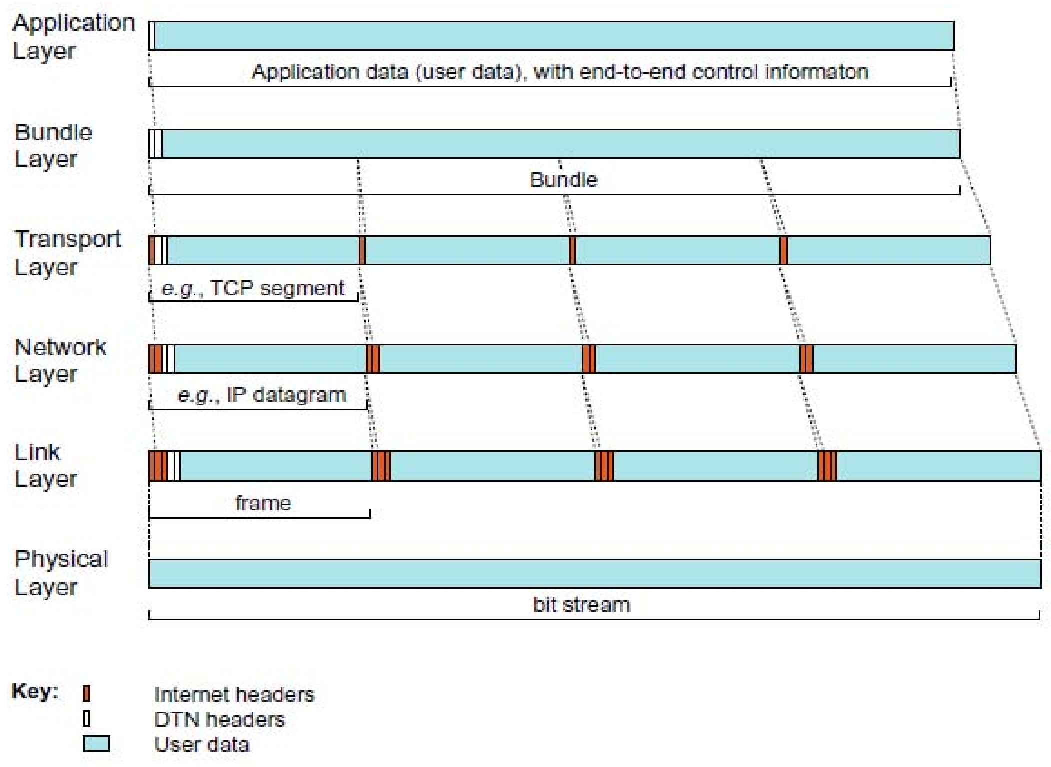

Figure 8: Encapsulation with the Bundle Layer [From 8]

In a DTN, a node is an entity with a bundle layer. Each node in a DTN can be a host, a router, a gateway, or some combination of the three depending on its configuration and networking capabilities. A host can send or receive bundles but does not participate in the network by forwarding bundles to other nodes. Routers forward bundles received from other nodes that are not addressed to it. Finally, gateways serve as a bridge between two different network regions using two different underlying protocols. Every type of node requires persistent storage in order to hold bundles that it has to send over an intermittent link, or that have been received and cannot be forwarded onward yet.

C. ROUTING IN A DELAY TOLERANT NETWORK

With a basic understanding of how a delay tolerant network operates and how it differs from a standard TCP/IP network in place, we can now delve into one of the major research areas in delay tolerant networking: routing. Routing in delay tolerant networks is a matter of getting the bundle created at the source, to the destination node. This is complicated by the fact that an end-to-end path may never exist, or may only exist in the time-varying connectivity graph. To get around this complication, DTNs use store-and-forward routing to achieve eventual delivery at the destination. With each node in the network assisting the source node as a forwarder this is reasonably trivial. However, a complication arises when we look queue sizes, network traffic load, and the life spans of the bundles in the network. While we evaluate the various protocols, we will use the metrics of delivery rate and delivery delay to assess the performance of the different protocols.

As previously discussed in Chapter I Pg. 3-6, routing strategies for DTNs fall into two broad categories each with different advantages and disadvantages. These are the single-copy and multi-copy strategies. The basic difference between the single and multi-copy ideas is the number of copies of the message that exist. In a single copy case, only one is ever present in the network saving storage, bandwidth, and the power used to transmit it. However, the routing decisions must be made carefully to avoid sending the only copy even farther from its destination. In a multi-copy case multiple copies are made and spread through the network as time progresses. This consumes much more storage space across the network, bandwidth usage is increased as copies are sent multiple times, and power usage increases. The benefit of the multiple copy case is that in most every case the delivery delay is decreased as only one copy needs to successfully reach the destination. In addition routing is not as complex a decision. With the multi-copy approach being much simpler to implement in the real world, we will focus our efforts in that regime.

1. Epidemic Routing

The simplest method of routing in a DTN is multi-copy epidemic routing [10]. Epidemic routing, like its name implies, moves the bundle through the network in much the same manner that a virus or contagion moves through a population of carriers.

When a source node creates a bundle, it will wait until it connects to another node. If that node does not already have a copy of the bundle locally, then the source node will forward the bundle to the new node while maintaining a copy itself. After this process, there are now two copies of the bundle in the network, and both nodes holding copies of the bundle will continue to forward it to other nodes as they connect through the network. It has been proven that epidemic routing will always have the bundle find the shortest possible to the destination, assuming infinite bandwidth and infinite queues [10]. This result is due to the fact that over the amount of time it takes to forward a bundle from the source to its destination, there are a discrete number of possible paths that the bundle can take. A copy of the bundle is given to every node the source encounters, and every node that the subsequent carriers connect to. After some amount of time, every node in the network will have a copy of this bundle, unless a node has been completely segmented from the network. If a node connects to the network, it will eventually obtain a copy of the bundle, and this includes the destination. Of all the possible paths to the destination one or more will be the shortest possible based on delivery delay or delivery rate. By ensuring that every node that contacts a carrier gets a copy, all possible paths are simultaneously explored. This ensures that the shortest path to the destination is always found.

This method of forwarding is simple, effective, and always finds the shortest path. It would seem that research into DTN routing is a solved problem; however, there is that assumption we previously mentioned. This assumption causes significant problems with epidemic routing. Epidemic routing only works effectively with infinite queue lengths, infinite bandwidth, and infinite storage at each node.

The reason for this is just as simple as the routing strategy itself. When the network has delivered the bundle to its destination, the other nodes have no way of knowing this and therefore hold on to their copies. With this in place, eventually every node in the network will have a copy of every bundle ever sent, except for the ones addressed to it. The bundles addressed to a node can be safely deleted once the application layer has consumed the data payload.

Having the destination send out “delivery successful” bundles, prompting the other nodes to discard the bundle, seems to be a solution to this issue; however, it only delays the requirement for infinite storage. While bundles can be arbitrarily large, the delivery bundles can be very small, just a header and a few bytes of data. Eventually, like the data bundles, these delivery successful bundles will reach every node, unless they are permanently partitioned from the network, and then be stored there indefinitely. If we put a lifespan on them, there is always a risk that every node will not be reached. If we allow them to live forever, we still need infinite storage to hold them all. This scheme will fill up the storage far slower than the data bundles, but it will still fill up the queues eventually.

With this problem present, attempts to simulate the performance of the epidemic routing scheme with limited storage have been proposed and some have seen success in various areas and network topologies.

2. Spray and Wait [5]

In between the pure single-copy approach and the multi-copy approach is a method known as bounded-multi-copy. In this approach the multi-copy ideas are used, but the number of bundle copies present in the network is limited. In order to minimize the transmissions that happen in a network, a set number of bundles are distributed to nodes by some heuristic. These “relay” nodes wait until they subsequently encounter the destination node in order to directly transmit sometime in the future. This approach can be useful in highly mobile and social networks where a node can be assumed to meet every other node eventually. Spray and wait has two major advantages. First, by setting a maximum number of bundle copies, the memory size at each node, and the amount of data in the network as a whole, is limited. Unless a node is adjacent to a very chatty neighbor, or just travels near many transmitting nodes, it is unlikely that it will be overwhelmed with traffic to forward. Second, by only transmitting during the initial spray phase, bandwidth is conserved. The only other time bundles will be sent is when they are being delivered to the destination. This conserves power, a positive effect in energy constrained devices such as sensors in a network. A disadvantage of this routing strategy is the assumption of random movement. Random movement is used in many simulations but is rarely seen in practice. If Spray and Wait is used and the nodes that receive a copy of the bundle never cross paths with the destination, then the system fails completely.

3. MaxProp and Diesel Net [4]

MaxProp is an experimental routing protocol developed by the University of Massachusetts at Amherst. MaxProp uses a multi-copy approach in addition to a variety of techniques to improve delivery chances and control previously delivered bundles in the network. MaxProp uses a sorted queue to store bundles at each node. This queue is sorted by the hop count for bundles with a hop count less than some user defined threshold, or by an estimated delivery cost for those above this threshold.

The first technique MaxProp uses to determine if it should forward a message is hop count (the number of times a bundle has been forwarded.) This is useful because new bundles in the network that have a long way to travel will necessarily have a high delivery path cost. This can push them back in the queue behind bundles that are closer to their destinations. In a large network this can come to a point where they never get a chance to propagate across the network. By moving bundles that have a hop count smaller than a set threshold to the front of the forwarding queue, MaxProp helps move these bundles through the network quickly. The bundles that are past this hop count threshold are forwarded based on the path cost calculations. This utilizes the fact that bundles with high hop counts may have been delivered elsewhere in the network already.

MaxProp determines delivery path cost by attempting to assign weights to the links in the network. To accomplish this each node maintains a data structure that keeps track of past contacts with other nodes in the network. MaxProp keeps track of past contacts because it uses the assumption that past contacts will be a good indicator of future contact opportunities. At network initialization each node’s probability of meeting any other node is 1/(|s| - 1), where |s| is the set of nodes in the network. So, in a five node network all nodes will have a probability of meeting any other node equal to 0.25. When a node meets another node in the network, its probability value is increased by one, and then all the values are renormalized so that they again sum to one. In our five node network, if one node encounters another, its value will increase to 1.25 while all other nodes stay at 0.25. These values are then renormalized, and this results in the value for our contacted node being 0.625 and all the others set to 0.125 In this way, nodes that are not encountered frequently eventually obtain lower values.

In Figure 9, we see an example of a simple four node network. Nodes A,B,C, and D are connected to each other by lines representing available connections. The path cost algorithm is used to make routing decisions by looking at all possible paths available to a bundle and then choosing the least expensive. The path cost for the path ABD is calculated by adding together the individual hop costs. The path A to B costs 1-(the value of B at A) or 1-0.5. The path from B to D costs 1-0.25. Together they give the path ABD a cost of 1.25. Calculating these costs gives an idea of which path is the most likely to result in delivery, in the shortest period of time.

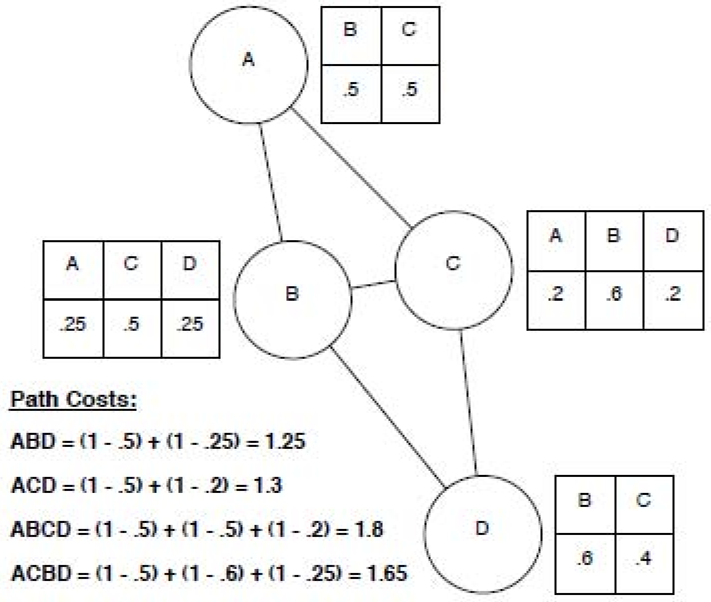
A
B C
.5 .5
B
A C D
.25 .5 .25
C
A B D
.2 .6 .2
D
B C
.6 .4
Path Costs:
ABD = (1 - .5) + (1 - .25) = 1.25
ACD = (1 - .5) + (1 - .2) = 1.3
ABCD = (1 - .5) + (1 - .5) + (1 - .2) = 1.8
ACBD = (1 - .5) + (1 - .6) + (1 - .25) = 1.65

Figure 9: MaxProp Path Cost Calculations [From 4]

Finally, MaxProp uses an Acknowledgement or Ack bundle to signal other nodes that a bundle has been successfully delivered and can be deleted from their queues. These Ack bundles are only 128 bits in size and contain the hash of the bundle contents. When they are received the node will delete the corresponding bundle freeing up space in the node’s memory and preventing the deleted bundle from being forwarded again. There are three situations in which MaxProp will delete a bundle:

1. A copy of message *m* has already been delivered to its destination.
2. No route with sufficient bandwidth will exist between the current node and the destination during the lifetime of message *m*.
3. No copy of *m* has been delivered but some copy of *m* will be delivered even in the current node drops its copy.

The system uses the Ack bundles to evaluate criterion number one, the path cost calculations for criterion two, and hop count as a weak indicator for criterion three.

When forwarding bundles, MaxProp uses four distinct steps to accomplish forwarding:

- First, the bundles contained at the local node that are destined for the connected node are forwarded.
- Second, routing information is passed between the nodes. This allows them to determine an estimate of delivery cost by using this information as path costs and a modified version of Dijkstra’s Algorithm.
- Third, Ack bundles are passed to help clear out buffers at on both sides of the connection.

- Finally, other bundles are exchanged. MaxProp uses hop count to initially favor new bundles up to a threshold that can be adjusted. Bundles that have been forwarded less than this threshold are given higher priority in the queue and sorted by hop count. Bundles above this threshold are forwarded based on the estimated path cost for delivery determined in the second step. This results in bundles that have a higher chance to be successfully delivered being forwarded first, once all the low hop count bundles have been sent.

The experiments conducted using MaxProp showed that it performed better than a previous work protocol called Drop Least Encountered, and Oracle based Dijkstra path calculation algorithm, and a random router that selects which bundles to forward randomly. In addition, it was used in a real DTN environment implemented on busses traveling around Amherst (DieselNet), allowing results to be obtained showing MaxProps suitability for use in varied DTN topologies.

4. PRoPHET [3]

PRoPHET is a probabilistic routing protocol for delay or disruption tolerant networks designed by the DTN Research Group. Many attempts at delay tolerant routing achieve good performance by relying on simulations using random walk or random mobility models. This works well; however, in reality mobile nodes in delay tolerant topologies will tend to prefer contacts within a subset of nodes in the network to others. It is somewhat intuitive to realize that this is the case. If I have a close circle of friends who are also nodes in the network, it is far more likely that I will be in range to contact one of them than any random stranger on the street. In another case, vehicle based networks using public transportation also show that nodes running at similar times will have a higher probability of meeting a node running a crossing route repeatedly than one with no route intersections. PRoPHET attempts to take advantage of this underlying pattern by keeping track of the history of connections a node makes, and using it to try and predict the potential for future meetings and eventually, a successful delivery by a node.

The way PRoPHET determines delivery probabilities is through the use of several equations. At initialization, a node knows only of itself. As it connects to other nodes in the network, it learns of those nodes and, transitively, the nodes with which they have connected to in the past. Before the equations can be fully understood, however, we must define several variables.

- **P_encounter** – is used to increase the delivery predictability for a node when the destination node is encountered. A larger value of P_encounter will increase the delivery predictability more quickly and fewer encounters will be necessary for the delivery predictability to reach a certain level.
- **P_first_threshold** – When nodes are disconnected for long periods of time, the P_values necessarily decay. This prevents a network where all nodes

have a delivery probability of (1-delta). When a nodes P_value has decayed below P_first_threshold then any subsequent encounters by that node will be considered as if the node had been encountered for the very first time.

- **P_encounter_first** – is used to set the delivery probability for nodes that have just met for the first time, or after a long separation where their P_values decayed below P_first_threshold. It is used to quickly raise the P_value to a number useful for subsequent calculations when a history of encounters does not exist.
- **Delta** – is a user defined variable present in the update calculation to ensure that delivery probabilities between nodes stay strictly less than 1. It should be set very small in order to not significantly impact the range of possible P_values.
- **Gamma –** is an aging constant set between 0 and 1 inclusive.
- **Kappa** – is the number of time units that have elapsed since two nodes have connected.
- **Beta –** is used as a scaling constant that controls the impact that the transitivity calculation will have on the P_value at a node. It can be set between 0 and 1 inclusive. With a setting of 0 PRoPHET will not use the transitivity calculation and as the setting approaches 1 PRoPHET will favor transitive routes more heavily.

From these variables, we can look at the equations that PRoPHET uses to update the P_values when two nodes connect. In the definition of these equations, P(X,Y) is probability of node X meeting node Y. The first is the basic update equation. It is used when two nodes meet, it is not their first meeting, and their P_values have not decayed below the P_first_threshold value. P_(A,B) in the equation is the probability of node A meeting node B.

P_(A,B)=P_(A,B)_old+(1-delta-P_(A,B)_old)*P_encounter

The second equation is the decay equation. This equation is calculated by a node when a connection is made and before the values are sent to the other node. This equation ensures that P_values are accurate representations of how recently a node has been in contact with others.

P_(A,B) = P_(A,B)_old * Gamma^Kappa

Finally, we can look at the transitivity equation. This equation allows us to determine if an intermediary node is likely to meet a third node in the future. For instance, if node A frequently encounters node B and never encounters node C. If node B frequently encounters both node A and Node C, then node B is probably a good node to forward bundles destined to node C.

P_(A,C) = MAX(P_(A,C)_old, P_(A,B) * P_(B,C) * beta)

The transitivity equation uses beta to limit the weight of the multi path probability to the destination. While it may be a very high probability, it still requires two hops thereby inducing the possibility of further delay.

Using these calculated P_values to forward bundles is at the core of the PRoPHET routing system, and it provides the user with seven possible forwarding strategies to use depending on network performance and topology. When looking at the comparisons in the forwarding strategies, node A is the node holding a bundle destined for node D, and node B is a node that has connected with node A.

- **GRTR** – Forward a bundle only if P(B,D) > P(A,D). When two nodes meet, the sender only forwards the bundle if the recipient node has a higher probability of meeting the destination node than the sender. The sender still retains a copy of the bundle, provided sufficient buffer space exists since a better node may be encountered in the future.

- **GTMX** – Forward only if P(B,D) > P(A,D) and NF < NF_max. This strategy is similar to GRTR, but it includes a Number of Forwards (NF) metric where the bundle is only forwarded a maximum number of hops away from the source node.

- **GTHR** - Forward only if P(B,D) > P(A,D) or P(B,D) > FORW_Thresh. FORW_Thresh is a threshold value where the bundle should always be forwarded, unless it is already in the buffer at the other node. This implements an effective epidemic routing amongst nodes with very high P_values.

- **GRTR+ -** Forward only if P_(B,D) > P_(A,D) and P_(B,D) > P_max, where P_max is the highest delivery probability that the message has been sent to so far. This strategy works like GRTR, but it limits the forwarding to only nodes with progressively higher P_values for its destination.

- **GTMX+** - This is a combination of GRTR+ and GTMX. It only forwards the bundle if P_(B,D) > P_(A,D), P_(B,D) > P_max, and NF < NF_max hold.
- **GRTRSort** – Sort the bundles in descending order of P(B,D) – P(A,D). Forward the bundle only if P(B,D) > P(A,D). This works like GRTR but because the bundles are sorted in descending order of the difference in the P_values. It moves the bundles, seeing the largest improvement in deliverability probability along first.
- **GRTRMax** – Sort bundles in descending order of P(B,D) and then forward only if P_(B,D) > P_(A,D). This works like GRTRSort but works on absolute delivery probability rather than trying to maximize the improvement for bundles.

Queue space is always at a premium in DTN topologies and PRoPHET provides a number of queue control options for the user to choose from.

- **FIFO** – First In First Out. The first bundle received is the first bundle evicted when space is needed.
- **MOFO** – Evict most forwarded first. This requires the node to keep track of the number of times that it has forwarded to a node to others. The bundle that has been forwarded the most is evicted first.

- **MOPR** – Evict most favorably forwarded first. Keep a variable FAV for each bundle in the queue. Each time a bundle is forwarded update FAV via the equation:

  FAV_new = FAV_old + (1 - FAV_old) * P

  where P is the P_value that the node the bundle is being forwarded to has for the destination node. The bundle with the highest FAV value is the first to be evicted from the queue.

- **Linear MOPR** – This is just like the MOPR scheme except that it uses the equation:

  FAV_new = FAV_old + p

- **SHLI** – Evict shortest lifetime first. All bundles in PRoPHET have a lifetime as described in RFC5050 [9]. Bundles whose lifetime is nearing its end will be dropped soon. This strategy evicts the bundle with the shortest lifetime remaining first.

- **LEPR** — Evict least probable first. Nodes are unlikely to deliver bundles for which they have a low P_value. This scheme drops the bundle that has the lowest probability of being successfully delivered by this node, and has been forwarded at least MF times. MF is a minimum number of times a bundle must have been forwarded before it can be dropped by this policy, if such a bundle exists.

With all of these strategies, PRoPHET is a highly adaptable and vastly configurable protocol that is suitable to just about any DTN topology provided the variables are configured correctly, and a forwarding strategy suitable for that network is selected.

D. BYTEWALLA DATA MULE SYSTEM [6]

The Bytewalla project connected two end points with mobile cellular phones as the go-betweens in a simple DTN topology.

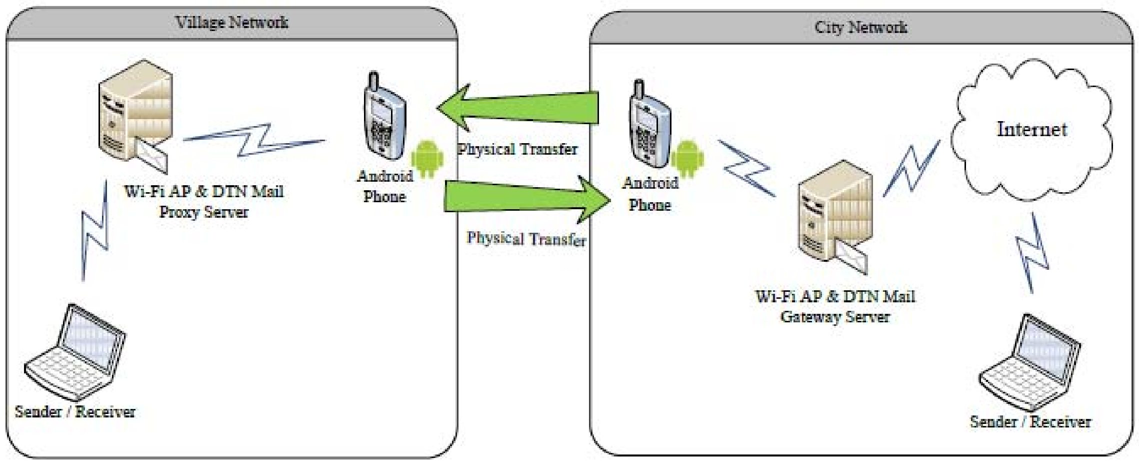

Figure 10: Bytewalla Architecture [From 11]

The goal was to effectively connect a remote village with a connected area, in this case a city, by using android phones given to people who regularly moved from the city to the village. This project leveraged the strengths of the DTN concept and moved it to a mobile phone. It stopped short of actually porting the DTN daemons onto the mobile phone, instead using the mobile phone as a wireless memory stick. The phones would connect wirelessly to a gateway at either end and exchange bundles as needed. While Bytewalla did offer DTN type functionality on a mobile phone, it proved unsuitable for our needs in this thesis. The phones were not capable of transferring bundles between them. They could only transfer bundles to a fixed access point at either the village or the city. Completely porting the DTN system onto a mobile platform still requires significant work and is left to future thesis projects.

E. DTN2

DTN2 is a reference implementation of the bundle protocol, and attempts to “clearly embody the components of the DTN architecture, while also providing a robust and flexible software framework for experimentation, extension, and real-world deployment.” [7] While it is developmental software, and still has some bugs in the code, it is the most stable example of working DTN available and is supported by the experts at the DTNRG. The current version of DTN2 available is DTN-2.7.0 and is available from sourceforge.com at [12]. Additionally, the user manual and documentation can be found at reference [13]. This representation will be used as a base reference for the rest of the experimentation.

F. SUMMARY

All of the technologies and routing concepts discussed in this chapter were developed and evaluated with the assumption that all the data (bundles) in the network is of equal worth. In reality, this is rarely the case. Spam messages flood our inboxes every day. Denial of service attacks flood networks with useless data and multiple demands for resources. In addition, what if, while the node was disconnected, some vital piece of data was injected into the network? MaxProp handles this to some extent by preferring to forward bundles that have only moved a few hops. PRoPHET has no utility to prefer bundles which contain high priority data. This means that they can get caught in a queue behind less valuable data. This thesis aims to implement a modified DTN routing system that maintains the performance gains made by PRoPHET and MaxProp while allowing preference for high priority or more valuable data bundles.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL METHODOLOGY

A. EXPERIMENT OBJECTIVES

It is the intention of this experiment to develop a routing scheme that will leverage the survivability and flexibility of DTNs while allowing for message priority to order messages being exchanged between nodes on the basis of the importance of their data contents. For this ordering, we leverage the priority fields already implemented in the bundle protocol header. This field provides three differing levels of priority and is contained in every bundle header even if it is not used.

B. THEORETICAL SYSTEM OPERATION

We start from a simplistic assumption that when a message is created by an application, the user will be prompted to give the message priority. This assumes that the users of the system are trustworthy and responsible individuals who will not abuse the system to obtain faster delivery times of their data bundles. We further assume that the semantics of the priority field are known and consistent throughout the network.

There is no difference in the creation of the bundle from the data besides the use of the priority field. The establishment of the channel between the nodes is also unchanged. The difference comes when the nodes exchange offer messages. Current routers like MaxProp and PRoPHET ignore the priority field; our routers will use it to sort the bundles prior to creating an offer message. These messages are created after the connection is established and the nodes decide which bundles in their queue to offer to the other node. Our routers will craft the offer message in such a way that all of the high-priority bundles are placed at the top of the list in a FIFO manner, then followed by the medium priority bundles, and finally by the low priority bundles. The node receiving the bundle offer then only has to start at the top of the list and request any bundle it currently does not have a local copy. Using this method, we ensure that bundles are sent in priority order. In this way, if to nodes experience a limited connection period where all bundles cannot be exchanged, the bundles transferred are higher priority than the bundles not transferred. To accomplish this we implemented our first forwarding strategy called COS, or Class Of Service. In this system we simply order the bundles in a nodes queue by their priority order. If two bundles have the same priority, then we will resort to FIFO ordering. This way when nodes connect the higher priority bundles are exchanged first.

In addition to the simple of exchange of bundles in priority order, we also implemented a second forwarding strategy that attempts to help limit queue size by restricting the forwarding of bundles based on the priority field. This forwarding scheme is called COSFLEX (Class of Service: Flexible forwarding). The expedited priority bundles will be forwarded using the GRTR method. In this case, if node is connected to a node that has a higher P_Value than the current node, it will be forwarded. This allows the bundles to travel unimpeded to the destination through nodes that have an ever increasing probability of meeting the destination node. Normal priority bundles will be filtered using the GTMX scheme. This scheme allows bundles to progress through the network through ever increasing probabilities of successful delivery up to a certain number of hops. If the NF or number of forwards has been met, the bundle will remain at that node permanently until it meets the destination node, is evicted, or expires. As an initial value we will set NF to be the ln(|s|), where |*s*| is the set of nodes included in the network. For the lowest level of message priority, we will again use the GTMX scheme, however, this time the NF will be set to $\mathrm{Log}_{10}(|s|)$. These values are just an initial starting point. They are based on the intuition that the path lengths from one node to another in a network are unlikely to be close to the size of the network. In this manner, we seek to limit the spread of lower priority bundles by limiting their path length to some non-linear function of network size. Using these two schemes for lower priority bundles will help reduce network traffic by restricting their forwarding to a set path length. This will reduce the number of copies in the network and create an effect similar to the Spray and Wait technique.

While using COS and COSFLEX we must not neglect the management of the queue itself. Even with the priority system and forwarding restrictions in place, under heavy traffic loads and long delays between connections, we expect that there will still be times when the queues grow too long for the storage allotted. In this case, some bundles will need to be deleted. To ensure that higher priority bundles are not deleted when lower priority bundles are still in the queue, we implemented a queuing strategy called Priority.

Inside of Priority, we identify candidates for eviction by examining priority values. When two nodes share the lowest priority, then we choose the oldest by using a FIFO sub criterion. When the eviction candidate is identified, this bundle will be removed to make room for the newer arriving bundle. In this manner, we preserve higher priority bundles, and when a bundle needs to be deleted it is the oldest bundle in the lowest priority present. This ensures that the chances it has been successfully forwarded in the past are as high as possible and no higher priority information is deleted.

C. TESTBED DESIGN

In order to evaluate our prioritized routing protocol, we perform several tests over several weeks. Before we begin testing, however, we have to set up a test bed. We will use a network of 40 to 50 highly mobile nodes. These nodes will be mobile phones carried by students during their daily activities. They will be set up in a similar manner to the Bytewalla data mule phones, except they will be able to transfer data between the two phones as they pass rather than just communicating with the endpoints. To communicate, they will use their onboard 802.11 connections in an AdHoc mode. We will use 40 to 50 mobile phones distributed amongst students in differing courses of study and differing progress through their respective curricula. This will ensure that phones are not continuously connected as groups of students move from class to class. Instead, phones will occasionally meet in a class but more often they will meet briefly in passing between class periods. The phones themselves will automatically generate between ten and fifty messages a day to random recipients in the network. This mirrors a Nielsen study of SMS usage, with the average users sending around ten per day and heavy users upwards of fifty. [14] Text messages contain only a small amount of data, so do not directly reflect the large bundles seen in a DTN, but this study of message rates provides us a data point to base our message generation scripts around. To allow us to experiment with class of service provided in the bundle protocol we will randomly assign messages a priority. Initially, we will use a distribution where 10 percent of messages will be branded with high priority, 60 percent of the messages will be medium or normal priority, and the remaining 30 percent will be the low or bulk priority.

D. TESTING

To test the performance of the system, we will first have to understand the baseline performance of each of the forwarding strategies. To do this, we will perform a number of baseline experiments using unmodified DTN2 and PRoPHET forwarding strategies. We will allow the network to run for a week in each case before the data is gathered and analyzed.

- First, we will run the network with flooding being the only forwarding strategy; however, the message priority will not be implemented to allow us to see an epidemic routing situation where high priority bundles can get stuck behind lower priority bundles in bundle queues.
- Second, we will run the network with our implementation of class of service, but still with

a GRTR routing scheme. This will allow us to see any improvements that CoS sorting of the message queues has provided us outside of our plans to implement varying forwarding strategies for each class of message.

- Third, we will run the network using the PRoPHET GRTR forwarding strategy and no message priority. This will give us the performance we should expect to maintain for the high priority / expedited class of service messages when our routing scheme depends on message priority.
- Fourth, we will run the network using the PRoPHET GRTR+ forwarding strategy and no message priority with the FN set to four. We determine this number by taking the natural logarithm of the network size. In this case ln(50) is 3.912, and we round it up to four. This will give us the performance we should expect to maintain for the medium priority / normal class of service messages when our routing scheme depends on message priority.
- Finally, we will run the network using the PRoPHET GRTR+ forwarding strategy and no message priority with the FN set to two. We determine this number by taking the logarithm (base 10) of the network size. In this case log(50) is 1.699, and we round it up to two. This will give us the performance we should expect to maintain for the low priority / bulk class of service messages when our routing scheme depends on message priority.

With these baseline tests complete, we will run our experiment using our new COS and COSFLEX routing strategies coupled with our Priority queue policy. We will also adjust the bundle size and data storage capacity of each node for subsequent tests. In this manner we can observe the network performance in a stressed and unstressed condition.

E. EXPECTED RESULTS

It is expected that in an unstressed condition the size of bundles inserted and the insertion rate will not overflow the allocated queue size, or will do so very rarely. Therefore, most bundles are eventually delivered, and few if any are dropped due to running out of storage space. In this case the network will see high delivery rates for all priorities. Using our COS forwarding strategy, expedited priority bundles see the shortest delivery delays. Using our COSFLEX forwarding strategy expedited priority bundles will see even shorter delivery delays since COSFLEX restricts other bundles progress through the network. Our COSFLEX routing scheme will stop the bundles of lower priority messages after four and two hops, respectively, for the normal and bulk class of service. This is expected to slow the spread of these bundles through the network resulting in longer delivery delays. However, in an unstressed condition eventual delivery is expected.

Under stressed conditions where the number or size of bundles outstrips the nodes capacity in its queue, we expect to see the delivery delays for all bundles in the baseline tests to remain about the same. However, when using our COS and COSFLEX strategies coupled with the Priority queuing strategy, bulk bundles will be dropped from the queues in order to make room for higher priority bundles. This results in bulk delivery rates dropping significantly. In addition, their delivery delays will rise quickly as more and more normal and expedited priority bundles clog the bandwidth during the limited connection periods. Normal priority bundles may see some increase in delivery delays and possibly delivery rates. This will depend on the severity of queue overloading and the number of expedited priority bundles in flight at that node during the test run. The expedited priority bundles are expected to see little change in their performance except under extreme conditions where the entire queue is filled with expedited priority messages. In this case however, we expect that the performance will be no worse than the performance of the PRoPHET router’s GRTR scheme with no CoS functionality available.

IV. EXPERIMENTAL RESULTS

In setting up our experiments, we deviated from our initial design profile. The areas where we altered the design will be addressed in this section. In addition, we will detail the actual experimental test bed we used, problems encountered during the setup and testing, the solutions that were implemented to solve these problems, and finally the experimental results. In addition, we will provide an analysis of the results in terms of our research questions.

A. EXPERIMENTAL DEVIATIONS

1. Cell Phones

The current implementation of DTN2 and PRoPHET are coded in the C and C++ programming languages. Unfortunately, getting the code to run natively on an Android phone or iPhone would involve developing or recoding the entire system in a language that is compatible with the phones. While certainly a worthwhile endeavor, it was beyond the scope of this thesis and is left to a future developer. The Bytewalla system was investigated as an option due to the fact it currently runs on an Android platform as an application. This showed promise initially; however, we discovered that the actual Bytewalla system is limited in that it cannot communicate between phones. It only functions from a phone to a fixed base station, thus limiting its application to the scenarios considered for this thesis.

Given these limitations, the decision was made to use desktop computers running the Ubuntu operating system instead of mobile phones. This allows us to control the movement pattern of the nodes by controlling the access of the DTN daemon to the Ethernet port that we will use to connect them.

2. Students as a Mobility Model

Using actual students to create the test bed would add an additional layer of complexity that we could not control. While the performance of the new routing system in such a scenario is an interesting question, a controlled environment for the proving of the system changes should be sought before it is introduced to a real challenged network. This will provide a controlled baseline for comparison. This thesis opts to provide that baseline rather than jumping to an actual implementation. A large experiment with student participation is deferred to future work.

B. EQUIPMENT USED

1. Computers

For the setup of our test bed, eight computers were used. Two computers were Dell Optiplex model 745 desktop computers. These were equipped with an Intel Core 2 Duo operating at 1.80 GHz, 2GB of RAM, a Western Digital 80Gb 7200 RPM SATA II internal HDD, and a Broadcom NetXtreme BCM5754 10/100/1000 base-T Ethernet controller. The other six computers were Dell Optiplex model GX620. They were equipped with an Intel Pentium 4 dual-core processor operating at 3.4 GHz, 2 GB of RAM, a Western Digital 80Gb

7200 RPM SATA II internal HDD, and a Broadcom NetXtreme BCM5751 10/100/1000 base-T Ethernet controller.

2. Hubs

For our test bed we needed to network the computers together, and to meet this requirement we used two network hubs. One was a four port Netgear model EN-104TP and the other an eight port Netgear dual speed hub model DS-108.

C. SOFTWARE USED

1. Operating System

The operating system installed on each computer in the test bed was Ubuntu version 10.04 (Lucid Lynx) [15]. Once downloaded, the operating system was installed natively on the desktop boxes with standard options. Once the operating system was installed and functioning properly, it was completely updated with the included automatic updating system to ensure that the system was as stable as it could be. When this was complete, the following packages and all of their dependencies were downloaded using the included synaptic package manager.

a. Autoconf
b. Build-essential
c. g++
d. libavahi-compact-libdnssd-dev
e. libdb4.6
f. libdb4.6-dev
g. libdb4.7
h. libexpat1-dev
i. libxerces-c2-dev
j. libxerces-c28

k. mono-1.0-devel
l. tcl8.4
m. tcl8.4-dev
n. tcllib
o. tclreadline
p. tclx8.4-dev
q. tclx8.4
r. zlib1g-dev

2. Database Storage

The database software used for the DTN2 storage system was Berkeley DB version 4.7 [16]. We used the file Berkeley DB-4.7.25NC.tar.gz. This option has no encryption capability, but since we are not going to be working with encrypted bundles it suits our needs. The software was downloaded and then extracted to the desktop. Once the file was extracted, a console window was opened and changed to the directory build_unix under the extracted file system. The following commands were then entered to install the database backend for DTN2:

a. ../dist/configure
b. make
c. make install

3. Oasys

The Oasys files needed for DTN2 were downloaded from the sourceforge.net [17]. The file oasys-1.4.0.tgz was downloaded and extracted to the desktop. Once extraction was complete, a directory was created named dtn2 and the extracted file was placed into it. Once in this directory, the name of the extracted directory was renamed from oasys-1.4.0 to oasys. This change prevents problems later on during the installation of DTN2. DTN2 will look for oasys files and will not find them if the directory is named oasys-1.4.0. Once the file was renamed oasys, it was installed by using a console window. We accomplished this by changing to the dtn2/oasys directory and entering the following commands:

a. ./build-configure.sh
b. CC=gcc CXX=g++ ./configure
c. make
d. sudo make install

4. DTN2

The software for DTN2 itself was downloaded from the same sourceforge site as the oasys software [18]. Once the files were downloaded, they were extracted to the desktop and then moved into the dtn2 directory created earlier. To install DTN2, change to the dtn2/dtn-2.7.0 directory and enter the following commands:

a. ./build-configure.sh
b. CC=gcc CXX=g++ ./configure –C
c. make
d. sudo make install

D. SYSTEM SETUP

1. Network Construction

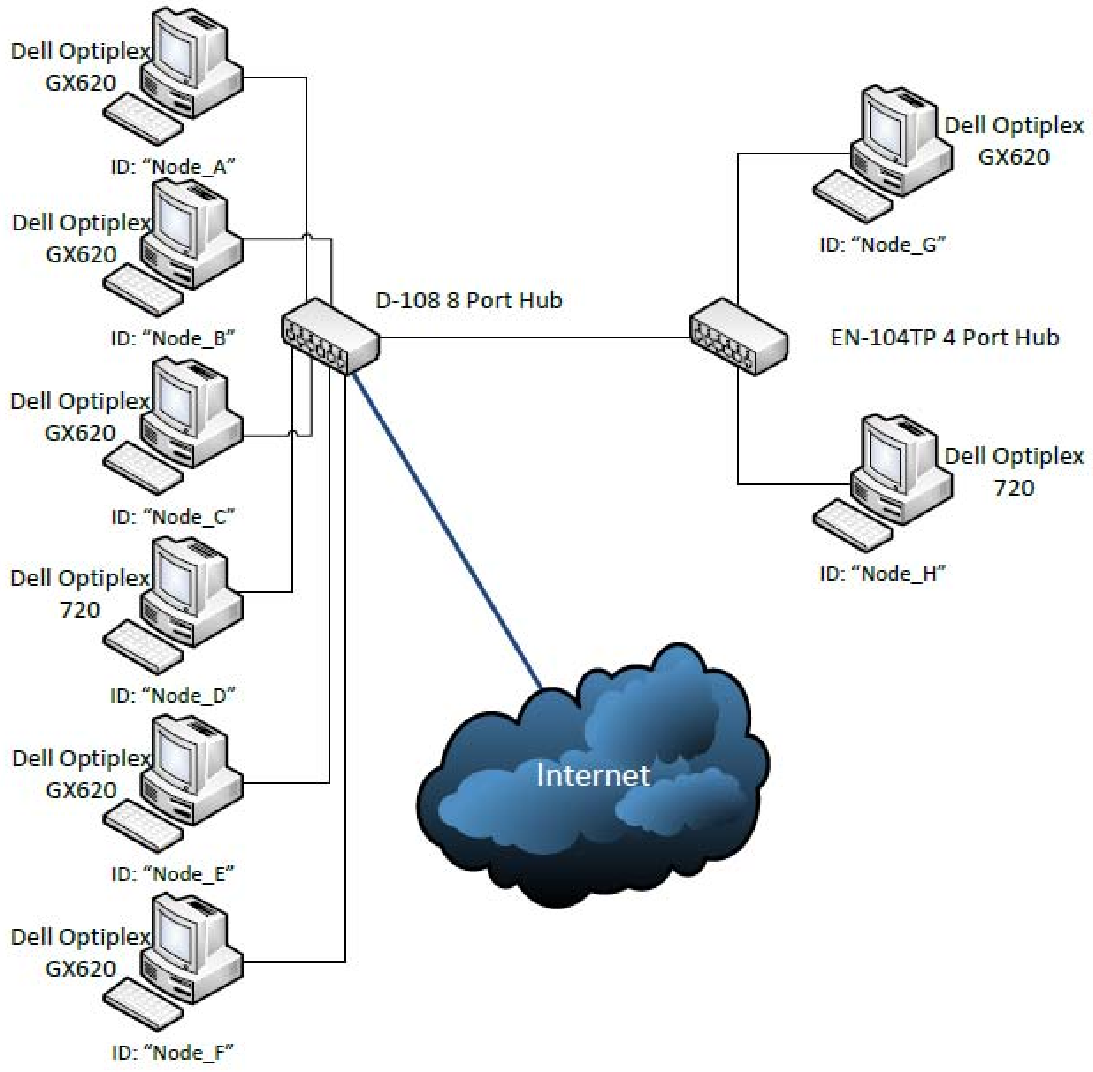

Figure 11: Network Construction Diagram

The eight desktop computers were connected with the two Netgear hubs as illustrated in Figure 11. The first four ports on the eight port hub were connected to nodes A through D. The fifth port was a crossover cable connecting the eight port and four port hub. Finally, the nodes E and

F were connected to the sixth and seventh ports respectively. On the four port hub, the first port connected the two hubs with the crossover cable, and the next two ports were taken by node G and H.

2. Bash Scripts

The computers used several scripts to emulate DTN nodes and cause disruptions to the network. Some are fairly simple, used to merely start the DTN daemon and establish a log file for data gathering, and the others are slightly more complex, and generate our bundles or disrupt the network. All of these scripts are included in the appendices.

a. Prophet.sh

This script starts the bundle daemon and establishes our log file. It uses two local variables in this script, datestring and newlog. The datestring is initialized by grabbing the current date and formatting it to suit our needs. newlog then uses datestring to create a filename for our log file. Then the script changes to the appropriate directory and starts the DTN daemon from the command line. There are several command line options when invoking this script.

- The “-c” option tells the daemon where to find the .conf file.
- The “-o” option is where we will put our output .log file.

- The “–l” option sets the log level. In this case we only get the information level log messages.
- The “-s” option sets the seed for the random number generator.
- The “-t” option is the tidy option. It starts the daemon with an empty database so that bundles from the last test run are deleted and don’t taint the results of subsequent runs.

b. Receive.sh

This script is run after the DTN daemon starts up. It simply starts up a process in the DTN daemon named test. This will be the endpoint that the bundles will be delivered to. Without this, the bundles would eventually reach the node and just sit in the queue without a process to consume them.

c. Bundle_send.sh

This script generates our bundles to be sent in the DTN network. First, it determine if a bundle should be sent. It does this by using a random number generator to generate a number between zero and ninety nine. If it is 49 or below, it sends a bundle, above, it does not. This results in the average number of bundles sent per hour being around thirty. With most hours seeing between 20 and 40 bundles sent.

In the next step, the script determines the destination for the bundle. This is accomplished by taking a random number and then, using the mod function, converts it to a number between 0 and 6. Each node in the network is given an unique identifier (Node A, Node B, etc), and using this number, a case statement then sets the variable destination to the appropriate id string for the dtnsend command line.

Then, it determines the priority the bundle. It does this in much the same way as determining the destination. It grabs a random number between zero and ninety nine, if the number is between zero and nine, it sets the variable priority to expedited. If the number is between ten and sixty nine priority is set to normal, and between seventy and ninety nine, set to bulk. Therefore, on average, 10% of the bundles generated are expedited traffic (i.e., with a high priority), 60% normal (medium priority), and the remaining 30% bulk (low priority).

Finally, it uses the dtnsend command that comes with the DTN2 package to send the bundle to the DTN daemon for delivery. dtnsend supports the following command line options.

The dtnsend command line has many options set enabling it to send a specifically crafted bundle.

- The “-P” option sets the priority for the bundle and is followed by the priority variable.
- The “-e” option sets the lifetime of the bundle. Currently it is set to 8640. This

is because our simulation is run at ten times speed and there are 86400 seconds in a day. The bundles will live for one day in the network before expiring.

- The “-W” option prevents the system from fragmenting the bundle if only a portion is received before disconnection.
- The “-s” option is the source node of the bundle.
- The “-d” option is the destination node for the bundle.
- The “-t” is the type of payload to be sent and is followed by the letters d, f, or m. The letter d sends the current date, the letter f refers to a file, and the letter m is a message.
- The “-p” option is the payload itself. With a date being sent, this field is not necessary. With a file, it is the complete directory path for the file to be sent. If the message option was selected, it is the message string to be delivered.

The script also echoes a summary message to the command line, so we can tell the script is operating correctly.

d. Disrupt.sh

This script creates the disruptions in our network that require the DTN to overcome. Initially, when the nodes are started, the network is completely shut down. When the script is started, it will sleep for a random amount of time between one and six minutes. This is to allow for nodes to come online at different times. After this initial period, it will come online and offline until the script is cancelled. The node will come online for a random time between one and two minutes, and then go offline for a random interval between three and four and one half minutes. This simulates a random mobility model where a node is likely to meet any other node, or subset of nodes, whenever it comes online. The weakness inherent in this approach is that a random network does not benefit from PRoHET's ability to anticipate future contacts by examining historical contacts. In the future continued work should focus on both a random mobility model and one where PRoPHET's history will benefit the routing and compare the two.

e. Scraper.py and CDF_Scraper.py

These two python scripts were created to read the large log files produced by the DTN daemon and gather the important data for analysis. Scraper.py reads the log files one at a time and outputs statistics for that node. CDF_Scraper.py takes all log files from a test run and gathers the delivery time data for use in creating the delivery time charts in the results section.

E. CHANGES TO THE PROPHET CODE

Unfortunately, the latest distribution of PRoPHET code had major bugs and was not functioning as expected. Many weeks were spent in debugging and fixing these problems. The difficulties encountered will be detailed later in the Problems Encountered section of this chapter. Most changes were made to allow the PRoPHET router to see the priority field in the bundles. Then use priority to make routing decisions with two new routing strategies and a new queuing policy. Once this was done, other changes were made to create specific log messages that could easily be parsed to gather data. All modified files can be found in Dif format within the appendices to this thesis. Next, we will briefly describe the changes made to each of these files.

1. dtn-2.7.0/servlib/prophet/FwdStrategy.h

In this file, the different forwarding strategies comparators are implemented. There are four areas inside the file where code was added. The first two additions are very similar. They just add the two new forwarding strategies to the existing enumeration and the forwarding strategy to string function (fs_to_str). The first addition is at line 44 and 45 and the second is at line 62 and 63.

The third addition starts at line 200. Here, we define our class of forwarding strategy comparator. This comparator works in conjunction with the deciders added later to effect a specific forwarding strategy. In this comparator, we first check the priority of the bundles. If they are the same, then we use simple FIFO ordering.

However, if the bundles are of differing priority, then we use the priority itself to compare and sort the bundles.

The final addition starts at line 285, here we add our new comparator to the strategy function. When called, this function looks at the forwarding strategy that is used and returns the appropriate comparator to implement it. Since both of the forwarding strategies we are implementing use the same comparator, we return the same comparator for both of them.

2. dtn-2.7.0/servlib/prophet/Decider.h

In Decider.h we implemented the decider function for our new COSFLEX forwarding strategy. In this file, the first addition starts at line 224. The decider we added is called FwdDeciderCOSFLEX, and this addition is the constructor and destructor for the decider used for our advanced forwarding strategy. The real meat of the decider function is in Decider.cc.

The final two additions are at line 292 and 323. These two additions add our two new forwarding strategies to the decider function. This function returns the appropriate decider based on the forwarding strategy selected. The COS (class of service) forwarding strategy simply uses the same decider as the GRTR family of forwarding strategies, so it is inserted at line 292. This is done because the GRTR decider uses a FIFO sorted list. Since we now have a list sorted by priority, the bundles offered to the peering node will be in priority order and FIFO inside the three different priorities. The COSFLEX decider uses its own specific decider and is included in the case statement at line 323.

3. dtn-2.7.0/servlib/prophet/Decider.cc

In decider.cc is the code for the new decider function. The first addition is in the include statements. Here we add the math.h library to allow us to use the logarithm functions, both the natural and base 10 versions. The next addition starts at line 201. This is the functionality of the DeciderCOSFLEX we created. Essentially, it uses a Boolean variable to make a decision to forward the bundle not. The variable we use is called num_fwd_ok. The first thing the decider checks is the priority of the bundle, this ensures that it uses the right criteria for forwarding. For high priority bundles, the num_fwd_ok is set to true since high priority bundles they are forwarded without any restrictions on the number of times they have been forwarded in the past. For medium and low priority bundles, we use the max_fwd number used by other forwarding strategies, except, in this case, the max_fwd variable is used to convey the number of nodes in the network. For medium priority bundles, we make sure that the bundle has been forwarded less than ln(max_fwd), and for low priority, the number of forwards cannot exceed log(max_fwd). In this manner, we limit the number of times a bundle can be forwarded to a number based on the size of the network. With this strategy, the number of forwards can be adapted by customizing the max_fwd variable in the .conf file.

4. dtn-2.7.0/servlib/prophet/QueuePolicy.h

In QueuePolicy.h we add a new queuing strategy to go along with our new forwarding strategies. The first two additions are at line 46 and 63. Much like the FwdStrategy.h file, these first two additions allow PRoPHET to recognize our queue policy by adding it to the enumeration and to the qp_to_str function.

The next addition starts at line 355. This is the constructor and destructor for our strategy. In the priority queue policy, the bundles are sorted by priority and then by FIFO within each priority class (expedited, normal, or bulk). This has the effect of ejecting the lowest priority bundle that has been around the longest, thus preserving high priority bundles in the queue until all other bundles have been dropped.

Our last addition begins at line 441 and adds our strategy to the policy function case statement. This function returns the proper queue policy comparator based on the queue policy selected.

5. dtn-2.7.0/servlib/prophet/Bundle.h

This file defines a façade interface between PRoPHET’s bundles and the bundle class used by the DTN daemon. Essentially, the router only needs a few of the pieces of metadata contained in the bundle to make a forwarding decision, so it only pulls those it needs into the router. To allow the router to see the priority, we added the code at line 48 to include the priority of the DTN bundle in the PRoPHET bundle when it is created.

6. dtn-2.7.0/servlib/prophet/BundleImpl.h

The BundleImpl.h file provides various constructors for the PRoPHET bundles; the code we added to this file is seen at lines 47, 62, 74, 90, 102, 117, 136, 150, 165, and 181. In all instances, we simply added the priority field from DTN to the PRoPHET bundle implementation. This includes all associated assessors, mutators, and destructors.

7. dtn-2.7.0/servlib/prophet/node.cc

In this file the only line that was changed was line 38. We changed the default value of Kappa from 100 to 3000. The reason this was necessary was that we were seeing seemingly random performance of the aging function when Kappa was set via the .conf file. When the DTN daemon was started it was unclear which value of Kappa would be used. After a recompile it would use the value included in the .conf file, but subsequent runs would use the default value of 100 contained in this file. To get around this, we opted to hardcode the value in this file. The problems stopped once the value was fixed in this file.

8. dtn-2.7.0/servlib/prophet/controller.cc

In this file the first change we made was at line 180. This line of code was eliminated because it was causing segmentation faults in a very specific situation involving the queuing policy. This situation develops when the bundle that has just arrived is the lowest priority bundle in the queue, and must be deleted. For some reason, this line was causing a segmentation fault by trying to access the sequence number of a bundle that had been dropped already.

Since this is just a log message that is not necessary for the correct function of the router, it was removed and the segmentation faults ceased.

9. dtn-2.7.0/servlib/prophet/repository.cc

The modification of code in the PRoPHET repository.cc file starts at line 99 in the add function. This code was changed to prevent the current_ variable, representing the size of memory currently occupied by bundles, from becoming negative and causing a node crash. A detailed explanation of this problem will be given in the Problems Encountered section. Further code was changed at line 170. This code modification was done to prevent the segmentation faults caused by the evict function attempting to delete a bundle and then calling the drop_bundle function to delete it a second time. Again, we defer a further explanation of the problem and its solution to the Problems Encountered section.

10. dtn-2.7.0/servlib/routing/ProphetBundle.h

In this file the added code was at line 105. This was added to the file to pull the priority field from the DTN bundle to the PRoPHET bundle.

11. dtn-2.7.0/servlib/routing/ProphetBundleCore.cc

The only code that was added to this file was added at line 187. This code was added to allow us to track bundle deletion from the queue, and to troubleshoot the performance of our new forwarding and queuing policies.

12. dtn-2.7.0/servlib/routing/ProphetRouter.cc

The first change to this file starts at line 148. This code was deleted because it effectively defeats the queuing policy. By asking the base class and then exiting if it returns false, the queuing policy will never be activated. The evict function only gets called when the queue is over its storage quota. By allowing the daemon to say no first, the queue will never be larger than the quota and PRoPHET’s evict() will never be called. By eliminating this code, the bundle is always accepted and if it goes over its size limit with the addition evict() will be called and handle it.

This created an unusual problem, and showed a potential reason why it was never allowed to be called. The problem that arose was that every time the evict function was called, a segmentation fault occurred. After extensive code walking and study, it became clear that there was a logical error in the evict function. The evict() function would delete the bundle from PRoPHET’s repository, and then call the delete_bundle() function to cause DTN to drop the bundle from its memory. The problem arose because the delete_bundle() function tries to delete the bundle from PRoPHET’s repository after it has been deleted by the evict() function, thus causing the segmentation fault. By removing the deletion command in the evict() function we prevented the segmentation faults, but then encountered a problem updating the size of the bundles currently in residence at the node. For some reason, the size of memory was being decremented twice for each deletion. To fix this, we deleted the line in the evict() function that decremented the size, as this was handled in the delete_bundle()

function also. The modified code has the evict() function used only to identify the bundle to drop, and then the delete_bundle() function is where the actual memory management happens. Follow up testing, by inserting bundles of varying priorities, showed that this had fixed the logical errors in the evict function and returned the system to a usable state.

The next change to this file was at line 218. This code was added to check that when a bundle arrives at the node it is checked for its source. If the source is an application, our dtnsend command is an application, then it prints the log message. If the source is anything else, such as an admin message or a bundle received from a peer node, it is not printed. This gives us an accurate count of how many bundles were injected into the network, at that specific node, during the test run.

13. dtn-2.7.0/servlib/cmd/ProphetCommand.cc

In this file, the changes made were at line 160 and line 193. These changes allow the parser that reads the .conf file to recognize our new routing and queuing policies. Otherwise, it would throw an error and exit every time it tried to read in the .conf file.

F. PROBLEMS ENCOUNTERED

Some of the problems encountered in implementing the system described above have been explained in the code changes. For instance, the unpredictability of the kappa variable and the accept bundle code fragment that prevented the PRoPHET evict() function from being called, have already been addressed. The major problem that keeps cropping up throughout the data gathering phase of this thesis is that of node stability

The problem of node stability arose on every test run of the system. One node seems to fail within the first hour of the run, and between four and six more will fail over the remaining seven hours. Initially, most of the nodes were failing due to a failed assert statement at line 190 in the handle_bundle_received() function of the ProphetRouter.cc file. This was tracked to the root cause; the PRoPHET router does not handle fragmented bundles correctly. When the PRoPHET router receives a fragmented bundle it comes from DTN’s internal fragmentation engine. Due to this, it enters a code segment to add the source of the fragment to the PRoPHET link list. However, the DTN fragmentation engine does not create a bundle with a link and so the assert fails and the node quits. With the random nature of segmentations in our test bed, we would frequently encounter a bundle that was fragmented due to a connection being severed while the bundle was being transmitted. To handle this occurrence, the DTN daemon performs reactive fragmentation by creating a bundle fragment that will be reassembled once the rest of the bundle is received. To enhance the node stability and fix this problem, we had to prevent the bundle daemon from fragmenting the bundle and then handing it to the PRoPHET router. This was accomplished by forcing the DTN daemon to not fragment bundles. To do this, we used the command param set reactive_frag_enabled false in the .conf file and added the “-W” option to the dtnsend command to create the bundles. The command prevents the bundle daemon from attempting to fragment the bundles, and the “-W” option for dtnsend command sets the do not fragment flag in the bundle metadata to true. This initially seemed to stabilize the network performance, however, the node failures continued.

Another problem encountered was the immediate failure of the nodes upon startup. This problem only happened when the nodes were allowed to connect with an empty queue, and with the queue size limited to below 2^{32} bytes. After extensive trouble shooting, the error was traced to a concurrency issue in the PRoPHET code.

The problem began when the local node started to communicate with a newly connected node. A trace of the log file while the log level was set to debug showed that when the PRoPHET node created a bundle, its representation was added to the PRoPHET repository almost immediately. However, when it was added to the repository its payload had not yet been written. As such, its payload size was zero. An examination of add() and del() functions in the Repository.cc file showed that the current_ variable representing the current size of all bundles at the node is updated as the bundles were added and deleted. The resulted in all locally generated bundles from PRoPHET being added with a size of zero, and then deleted with their proper payload size correctly represented. This resulted in the current_ variable eventually becoming negative. However, the variable is an unsigned 32 bit integer. When it becomes negative, the PRoPHET router saw it as a very large unsigned integer. This then triggered an attempt to purge bundles via the evict() function since current_ is larger than the max allowed storage size. PRoPHET would then delete bundles in order to bring the memory size down to below the maximum allowed size. Unfortunately, there are not enough bundles in the queue to accomplish this, and the node will fail due to a segmentation fault while it tries.

This was fixed by not using the current_ variable as originally coded. Instead, add() references the total_size data member of DTN’s own BundleStore class to determine if the queue size has exceeded the maximum. This change was verified correct by a hand trace of the code and resulting memory size variables. The only issue that could crop up is a two to three millisecond delay between the update of the current_ variable in PRopHET and the total_size variable inside DTN2. This is due to DTN2 making more checks about a bundle before deleting it. PRoPHET, on the other hand, merely deleted a representation of the original bundle. This delay results in a slight overestimate of the memory size for a small period. If a bundle were to arrive inside this small window and push the size over the maximum memory allowed, it could result in an erroneous eviction of a bundle.

Despite all these efforts, every node has failed due to a segmentation fault. Each segmentation fault seems to come from an attempt to delete a bundle or search for a bundle after it has been deleted. All segmentation faults observed originated from a call in the PRoPHET router. We suspect this is due to the PRoPHET router being authored in 2007, whereas the DTN2 code has been updated and changed with the current iteration being released in 2010. This evolution of the DTN code while the PRoPHET remained fixed, seems to have destabilized the system when the PRoPHET router is used. We would have liked to test the system with a stable base, however, the time to track down every segmentation fault was not available, and the rewriting of the PRoPHET code was beyond the scope of this thesis.

For this reason, each of the data test runs will detail which node failed at what time. Taking this into consideration, it will skew our test results but we should still be able to examine the delivery rate and time for bundles. With PRoPHET forwarding in use, the bundles created for the failed nodes will eventually age out. In addition, the bundles destined for failed nodes will not be forwarded to peer nodes as the p_values for the failed nodes will continually decrease and eventually be dropped. It will create a larger average queue length at a specific node, longer delivery times for a bundle since fewer nodes will be available to assist in eventual delivery, and will remove the benefit of the COSFLEX forwarding strategy since the size of the network is decreasing. This being said, we should still be able to use the test bed to evaluate the main research question of this thesis, that is, what kind of class of service differentiation we can make in a DTN environment,

A secondary problem that cropped up early on in the experimentation phase and was eventually resolved, was a problem getting two nodes to connect.

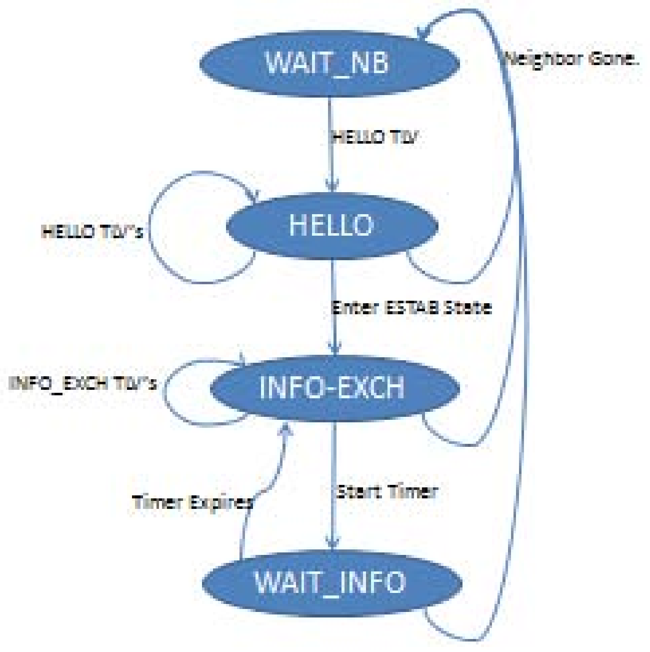

Figure 12: PRoPHET High Level State Diagram

When two nodes are not connected, they are both in the WAIT_NB state and awaiting a hello message from a neighbor while sending out their own periodic hello messages. Once a hello message is received, they enter the HELLO state and begin the setup of the connection between them. Once a setup is completed, both nodes enter the INFO_EXCH phase and transmit bundles that are selected for forwarding. When all bundles have been exchanged, if the connection is still up, they then set a timer and enter the WAIT_INFO phase. Once this timer expires, they will restart the INFO_EXCH phase. This allows the nodes to update the P-value multiple times over very long connection periods.

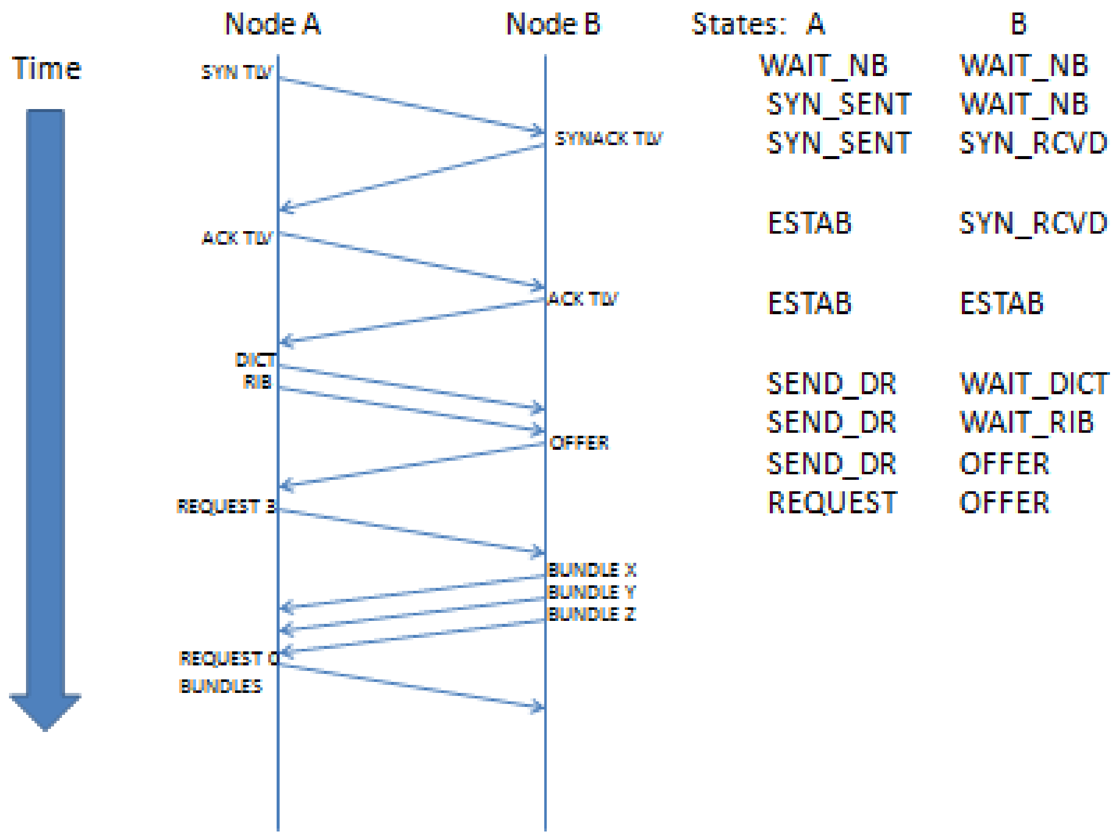

Figure 13: PRoPHET Connection Timing Diagram

Inside the HELLO phase, many bundles are exchanged to set up the connection and ensure that a neighbor is who it claims to be. When a node receives a HELLO message, it responds by sending a SYN message to the other node, and enters the SYN_SENT state. In a perfect world, the distant node receives the SYN message and enters the SYN_RCVD state. The distant node will then respond with a SYN_ACK message. The local node receives the SYN_ACK message and responds with an ACK message. The local node then enters the ESTAB state where it can proceed to the information exchange phase of the encounter. The distant node will also enter the ESTAB phase once it receives the ACK message.

The problem was, the links between the nodes would be up, but one node would stop in the SYN_SENT state and the other would stop in the SYN_RECD state. In this state, they never made a connection, and were unable to exchange bundles. This was troubleshot extensively over the course of several weeks, even enlisting the help of the DTNRG mailing lists. However, it was not until examining the log files of the DTN nodes in question that the solution presented itself. The problem was with time.

When two nodes connect, they send several bundles in order to complete the HELLO phase and enter the INFO_EXCH phase. These bundles only have a lifespan of forty seconds. So, if a node connects to another node whose system clock is more than forty seconds different from its own, the node that is behind will see bundles arriving that have a creation time in the future. For DTN this is not a problem, and it processes the bundle normally. The problem is at the other node. If a node sees a bundle arriving that was created more than 40 seconds ago it immediately drops it. This immediate expiration was stopping the nodes from finishing the connection process and exchanging data. In our case, the SYN_ACK message was being immediately dropped by the receiving node before it could be processed due to the short lifespan and drifting internal clocks. This might not seem a severe proble; however, the clock drift on our test bed required syncing the clocks between all eight computers about once every two days in order to keep them within forty seconds of each other. This 40 second lifetime was not mentioned in any literature I could find and had to be determined by observing DTN2 logfiles. Given the delays inherent in DTN operation it might be more useful to make this value closer to the one hour bundle life that DTN2 uses. This would require less frequent time syncs and be more tolerant of long delay links. There would be a slight increase in storage at the nodes but PRoPHET admin bundles are rarely larger than 100KB and should not negatively affect storage capacity with a one hour lifetime.

G. EXPERIMENTAL RESULTS

This section will detail our experimental results. Any deviations that happened during each run will be noted in each section.

1. Baseline Profiles

In the baseline profile, the network was run three times using the PRoPHET GRTR forwarding strategy and the FIFO queue policy. For all three runs, the bundles were 1MB in size. The bundles were generated by a script that created between twenty and forty bundles per hour. The size of the storage queue for the first run was 10GB. This gave a storage capability of ten thousand bundles. The second run reduced the storage to 1GB, giving a storage capacity of one thousand bundles. The final run reduced the storage yet again to only 100MB, resulting in a maximum queue length of a mere one hundred bundles.

During all three runs, the connection times were short when compared to the amount of data to be sent during each connection made. Based on our maximum possible connection time the number of bundles that could be transferred in a connection was between 1 and 150. This is the type of situation where our priority based routing system should shine. In the tables presented for each data run, we show a numerical summary of the test run. The first part of the table shows the lifetimes of the nodes in DD:HH:MM format and the numbers of bundles injected into the network broken down by priority. The second part of the table shows the bundles that were successfully delivered to each node by priority. Finally, we show the average delivery times required for each bundle seen by each node sorted by the bundle priority and the average queue length. On the right hand side of the table, we have summary numbers for the whole network.

Base run large Queue	A	B	C	D	E	F	G	H
Lifetime DD:HH:MM	1:10:10	1:00:12	1:00:37	1:00:32	0:05:44	1:10:03	1:10:03	1:00:05
High Injected	115	63	69	70	21	86	91	56
Med Injected	581	465	428	416	105	618	590	436
Low Injected	326	203	213	222	52	286	307	228

Total injected
571
3639
1837

High Delivered	36	15	12	6	0	4	11	6
Med Delivered	254	101	79	52	0	13	64	37
Low Delivered	144	71	53	36	0	6	30	16

Total Delivered
90
600
356

High Del Time	3644	2778	1482	1476	0	2370	4137	936
Med Del Time	3444	2502	2472	1040	0	1862	4621	1820
Low Del Time	3887	2671	2895	1204	0	2382	4489	1878

Avg Delivery Times (Sec)
2102.9
2220.1
2425.8

Avg Queue Length	672	476	409	418	74	464	505	455

Avg Queue Size (bundles)
434.13

Table 1.   10GB Queue Run using GRTR and FIFO

Examining the data for the first test (Table 1), we can see that all the nodes lived for approximately one day, except one that lived for only five hours. The bundles injected are given lifetimes of one day, and will expire if not delivered in that time. Since queue sizes were not an issue, the reason these nodes died at around a day of runtime was the segmentation faults caused by the handle_bundle_expired function in PRoPHET. Two nodes in the run survived, and were manually stopped after the other six nodes crashed. It is unclear why these nodes were able to handle the expiration of bundles correctly while the other six were not. This question is left for future work by other researchers.

Base run Med Queue	A	B	C	D	E	F	G	H
Lifetime DD:HH:MM	1:00:06	1:00:28	1:00:30	1:00:05	8:00:37	1:00:18	8:00:15	0:23:47
High Injected	68	79	82	69	574	58	557	77
Med Injected	468	451	438	449	3428	413	3483	421
Low Injected	222	196	201	206	1656	225	1667	207
High Delivered	23	5	10	0	55	5	5	2
Med Delivered	146	63	77	10	304	32	29	11
Low Delivered	77	37	40	1	151	24	13	6
High Del Time	2756	1245	1982	0	5908	3467	3305	2945
Med Del Time	3377	2782	2231	971	6044	3777	2553	3865
Low Del Time	3846	3132	2683	153	5957	3567	2413	3022
Avg Queue Length	637	448	493	348	702	446	669	425

Total injected
1564
9551
4580

Total Delivered
105
672
349

Avg Delivery Times Sec)
2701
3200
3096.6

Avg Queue Size (bundles)
521

Table 2. 1GB Queue Run Using GRTR and FIFO

In the second run (Table 2), the queue size was reduced from 10 GB to 1 GB. This reduced the effective storage capacity from 10,000 bundles to only 1,000 bundles. In a similar fashion to the first run, six nodes failed at near the one day point due to the same bundle expiration problem. Also similar to the first run, two nodes managed to survive beyond one day. The reduction in queue size performed in this run had no affect on the network. All nodes failed, or were manually stopped, long before reaching the 1000 bundle queue size limit.

Base run Small Queue	A	B	C	D	E	F	G	H
Lifetime DD:HH:MM	0:21:57	0:03:40	1:02:22	0:07:43	0:03:08	0:06:04	0:07:25	0:04:19
High Injected	54	10	84	28	13	25	21	9
Med Injected	399	55	460	133	67	115	147	73
Low Injected	217	34	237	71	17	54	61	39

Total injected
1564
9551
4580

High Delivered	2	1	1	5	0	1	2	0
Med Delivered	13	3	19	14	5	10	5	0
Low Delivered	10	1	9	6	4	6	1	0

Total Delivered
12
69
37

High Del Time	1495	252	1054	863	0	991	537	0
Med Del Time	434	654	570	531	194	455	1012	0
Low Del Time	1289	278	786	166	188	277	16	0

Avg Delivery Times (Sec)
649
481.25
375

Avg Queue Length	96	70	93	101	60	93	110	55

Avg Queue Size (bundles)
84.75

Table 3. 100MB Queue Run Using GRTR and FIFO

In the final run (Table 3) the lifetimes of the nodes in the network are greatly shortened. Despite our best efforts to fix and eliminate them, the faults in the evict function were never adequately removed to allow the queuing strategy to be executed without crashing the node. This causes the nodes to die in the amount of time it takes them to fill up their queues. For some nodes that manage to connect with another and exchange bundles, this is longer than others who do not, but in similar fashion to the other runs six nodes died when their queues filled up in between three and eight hours and two managed to survive for nearly a whole day without crashing. Looking at the bundles delivered in this run, it appears that most nodes made between 0 and 2 connections before crashing.

In Figure 14, 15, and 16 we can see the percentages of the bundles that were inserted with each priority, and the percentages of bundles delivered with each priority for each of the queue sizes. The blue columns represent the percentage of the bundles inserted with a specific priority. The red columns show the percentage of bundles that were delivered with a certain priority.

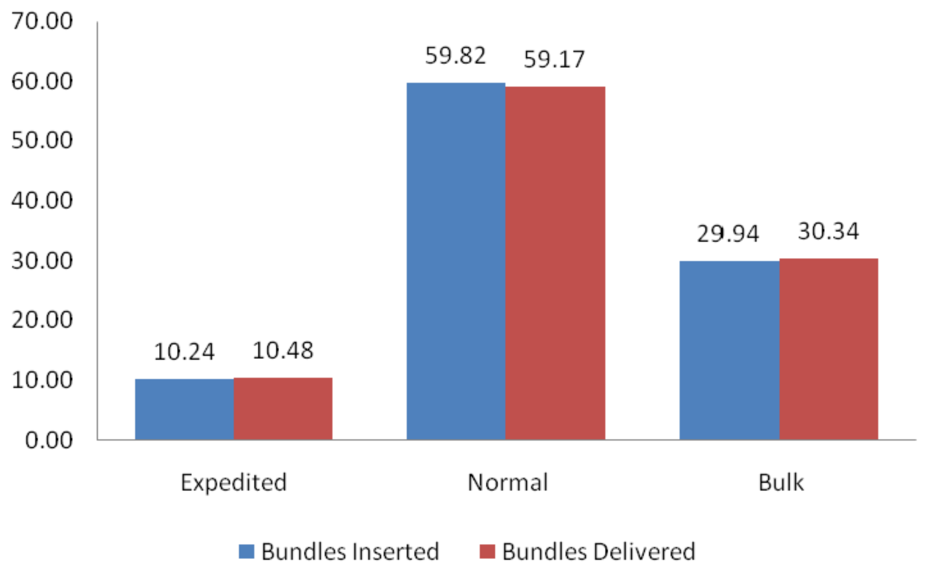

Figure 14: Insertion and Delivery Percentages for the 10GB Queue Test Run Using GRTR and FIFO

It is no surprise that across all three test runs we see no difference in the insertion and delivery percentages. The GRTR forwarding strategy uses a FIFO policy with some consideration for meetings the node has made in the past, but ignores priority completely. Since we are using a forwarding strategy that does not consider priority when making decisions, it makes sense that the delivery of bundles is in proportion to their insertion rates.

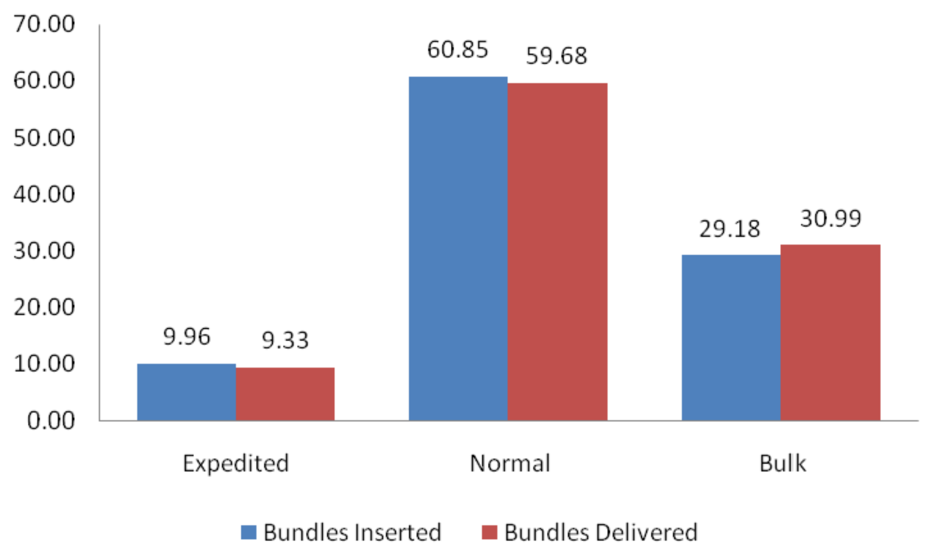

Figure 15: Insertion and Delivery Percentages for the 1GB Queue Test Run Using GRTR and FIFO

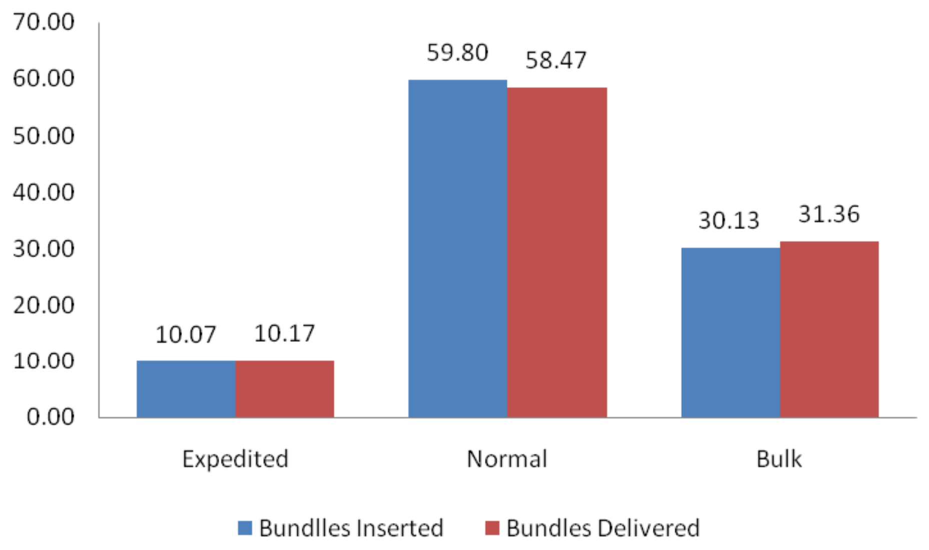

Figure 16: Insertion and Delivery Percentages for the 100MB Queue Test Run Using GRTR and FIFO

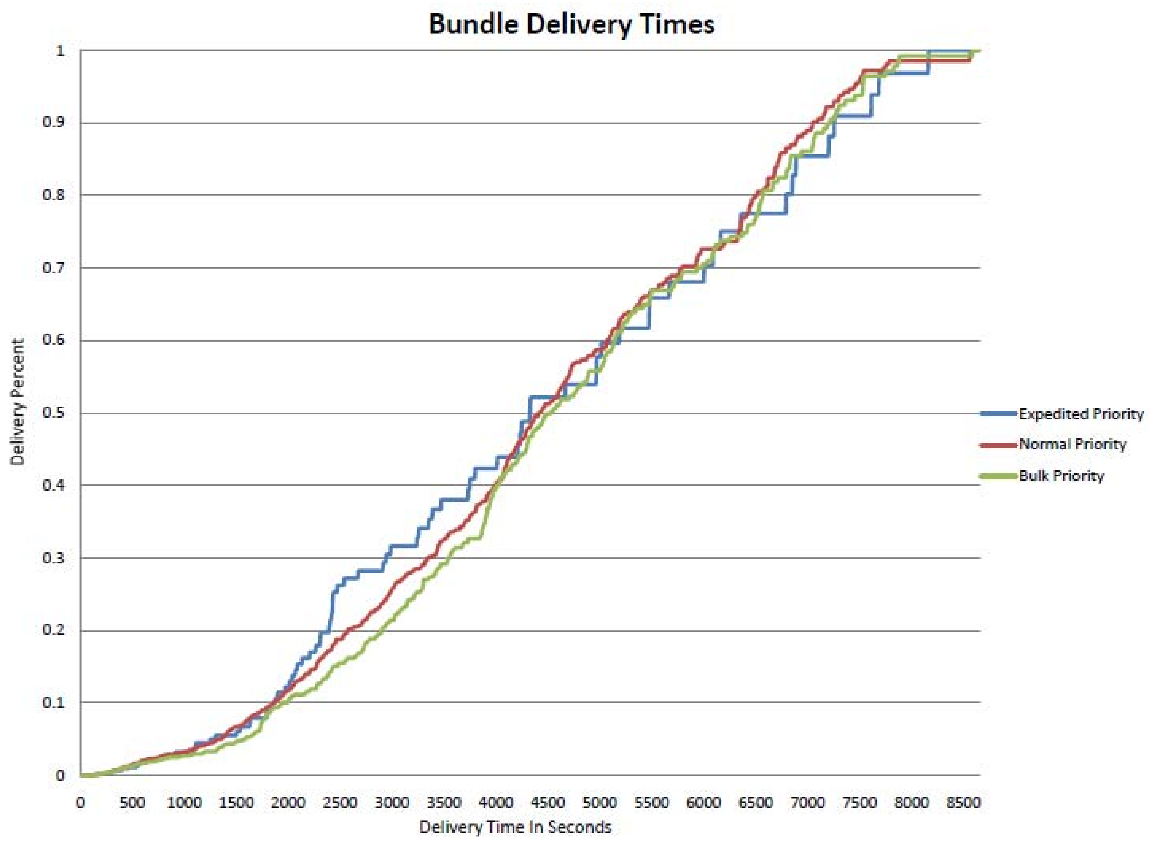

Figure 17: CDF of Delivery Times for the 10GB Queue Test Run Using GRTR and FIFO

In Figure 17, we see the cumulative distribution of bundle delivery times for the baseline test with a large queue. The X axis is the delivery time in seconds, and the Y axis is the percentage of bundles that were delivered. Not unexpectedly, the three bundle priorities are relatively close together in terms of delivery time. In Figures 18 and 19 we see this trend continue. If test runs had been conducted over periods longer than a day, several hours in the case of the 100MB queue test, we would expect to see graphs with even less deviation between the three bundle priorities.

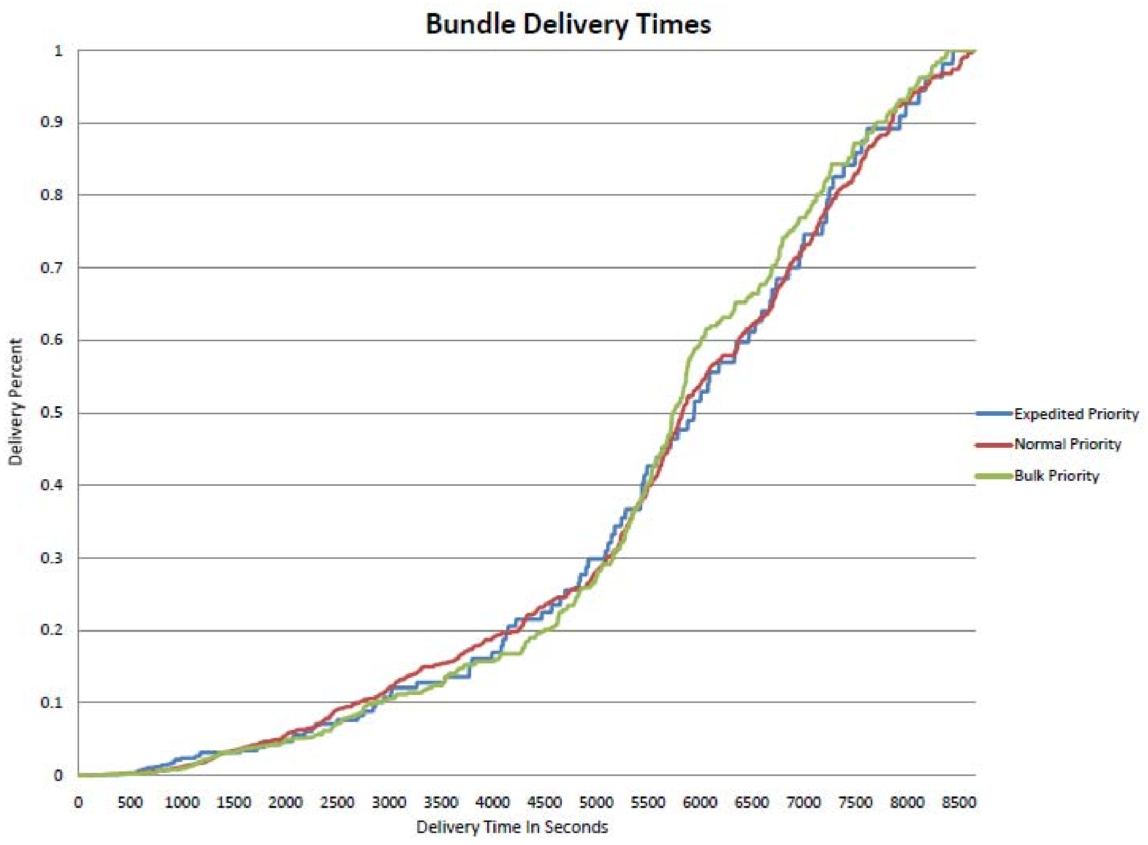

Figure 18: CDF of Delivery Times for the 1GB Queue Test Run Using GRTR and FIFO

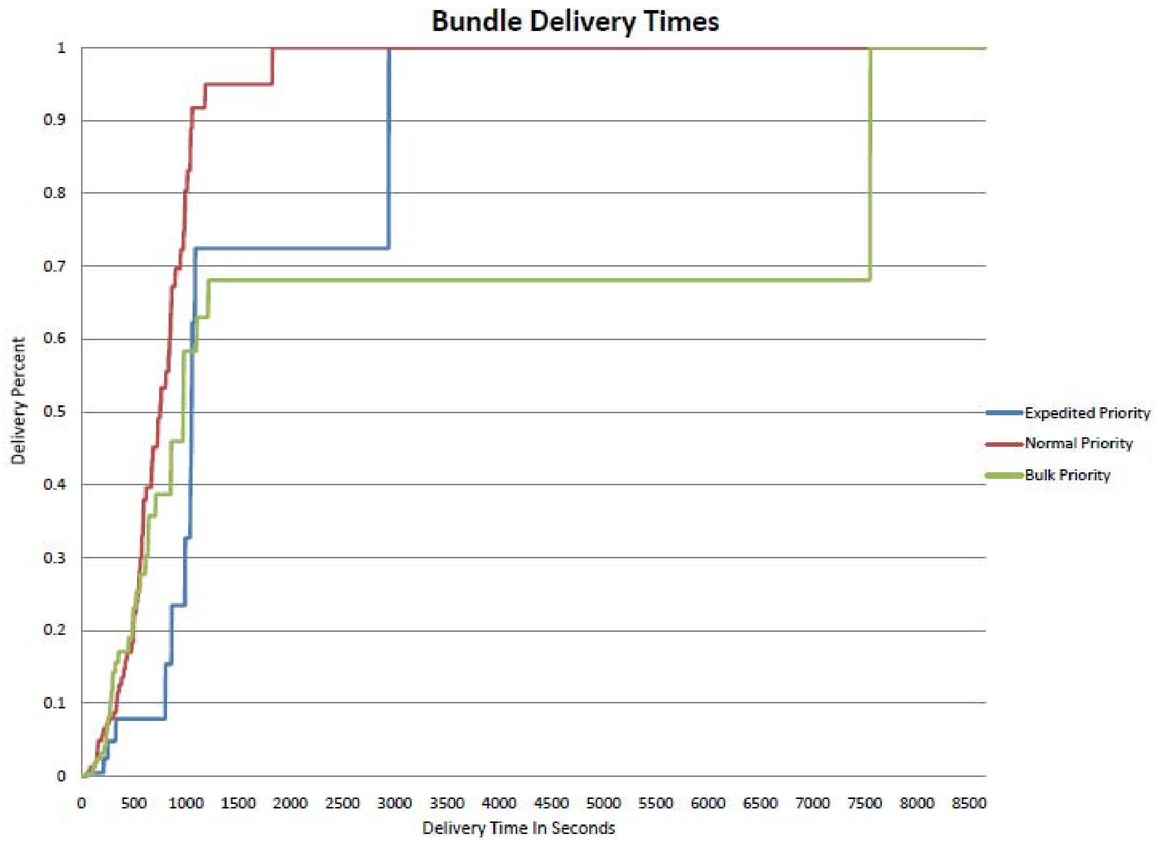

Figure 19: CDF of Delivery Times for the 100MB Queue Test Run Using GRTR and FIFO

In Figure 19, we see a very different graph from the results in Figures 17 and 18. This is due to the very small dataset provided by the short run times (hours vs. days) in the small queue test run. With the nodes failing in such a short time, the data appears to show a large jump in performance; however, this is just an artifact of the short node lifetime. This will present itself in all the 100MB queue tests we will examine.

Overall, in the baseline testing, we saw results that were as expected. All three data priorities achieved delivery rates equivalent to their insertion rates and delivery times that were approximately equal.

2. Experimental Profiles

Now we move on to evaluating the same three scenarios as the baseline test. Instead of using GRTR and FIFO, however, we used our COS and COSFLEX strategies coupled with our Priority queuing system. Similar to the baseline profiles, each experimental setup was run three times with the queue size varying from 10GB to 100MB. This required a total of six runs, three for the COS forwarding strategy and three for the COSFLEX forwarding strategy. In all runs we substituted the Priority queuing strategy for the FIFO queuing strategy that was used for the baseline tests. This was to ensure that both the forwarding strategy and the queuing policy were priority sensitive.

a. COS Forwarding Strategy

In the 10GB run, using the COS forwarding strategy, we encountered the same problem that we saw with the baseline testing. Six of the nodes failed around the one day mark when bundles started expiring, and two lived on. It is interesting to node at this point that the nodes that are continuing to function are not the same two from run to run. This seems to indicate that there is something in the randomness of the network contributing to the nodes running correctly. Unfortunately, time did not permit further investigation of this phenomenon.

COS run large Queue	A	B	C	D	E	F	G	H
Life DD:HH:MM	1:01:41	1:06:07	1:00:48	4:08:37	1:01:12	1:00:34	1:01:23	4:08:16
High Injected	76	80	73	318	76	68	88	336
Med Injected	463	575	428	1868	438	459	439	1815
Low Injected	222	262	229	918	202	234	201	920

Total injected
1115
6485
3188

High Delivered	53	61	29	72	27	20	20	23
Med Delivered	215	161	123	286	25	41	23	104
Low Delivered	67	21	15	112	3	1	7	16

Total Delivered
305
978
242

High Del Time	1931	1674	2145	1651	3707	1716	2178	3177
Med Del Time	2683	2282	3056	3004	4432	3045	1967	5122
Low Del Time	2754	2296	1580	3669	288	299	3365	5047

Avg Delivery Times (Sec)
2272.4
3198.9
2412.3

Avg Queue Length	573	458	512	751	460	403	487	625

Avg Queue Size (bundles)
533.63

Table 4. 10GB Queue Run Using COS and Priority

Examining the data in Table 4, especially the number of bundles delivered in this run, validates the initial intuition that adding the priority to the bundle forwarding strategy increases delivery for higher priority bundles. This can be seen by simply looking at the total bundles delivered columns on the right of the table. More expedited priority bundles were delivered than low priority bundles. Since expedited bundles are only created 10 percent of the time and bulk priority bundles are created 30 percent of the time, this initial observation shows promise.

COS run med Queue	A	B	C	D	E	F	G	H
Life DD:HH:MM	1:00:42	1:04:38	1:03:46	1:00:48	1:00:26	1:00:13	1:05:33	1:03:33
High Injected	77	82	82	71	86	72	90	61
Med Injected	432	538	533	456	443	441	533	531
Low Injected	231	240	222	234	228	212	284	241
High Delivered	51	41	38	35	11	3	11	9
Med Delivered	139	103	131	98	12	21	13	12
Low Delivered	53	28	35	22	0	9	2	3
High Del Time	2269	2428	1558	2764	3661	459	4354	2065
Med Del Time	2781	2750	2469	2979	5787	561	3822	1043
Low Del Time	3020	2997	2252	3313	0	917	5152	461
Avg Queue Length	594	657	654	480	486	409	499	433

Total injected
621
3907
1892

Total Delivered
199
529
152

Avg Delivery Times (Sec)
2444.8
2774
2264

Avg Queue Size (bundles)
526.5

Table 5.	1GB Queue Run Using COS and Priority

In the medium queue size run, the promising results from the first run appears a second time. This reinforces the confirmation of the successful use of the priority value to help delivery. Like the baseline runs, decreasing the queue size did not have an effect on the functioning of the nodes during the test. They crashed when the bundles began to expire; long before the queue became full and bundles needed to be evicted.

COS run small Queue	A	B	C	D	E	F	G	H
Life DD:HH:MM	0:05:38	0:06:44	0:15:54	0:06:35	0:04:15	0:13:24	0:05:25	0:03:18
High Injected	15	24	44	28	14	38	16	14
Med Injected	106	117	294	115	77	249	98	67
Low Injected	61	52	139	71	35	122	57	21

Total injected
193
1123
558

High Delivered	2	1	1	2	1	1	1	1
Med Delivered	6	20	4	15	5	14	9	5
Low Delivered	0	4	2	6	1	0	5	3

Total Delivered
10
78
21

High Del Time	44	184	301	351	152	125	232	116
Med Del Time	278	363	2212	477	229	383	498	97
Low Del Time	0	163	125	760	203	0	499	323

Avg Delivery Times (Sec)
188.13
567.13
259.13

Avg Queue Length	88	87	86	80	74	101	75	87

Avg Queue Size (bundles)
84.75

Table 6. 100MB Queue Run Using COS and Priority

The results of the small queue run using the new forwarding strategy initially appears to have gained no benefit from the new forwarding strategy. This is true; however, it is due mostly to the brevity of the test. Again six nodes crashed nearly immediately upon reaching their queue size limit, and two lived on for another ten hours before being stopped. When the tests are started, the queues are empty and bundles are generated to fill them. If a node connects to another node during this time, it is likely that no expedited bundles even exist in the queue yet, or if they do, there are very few of them. In this situation, the bulk priority bundles are more likely to be successfully forwarded. As the queues fill up with expedited and normal priority bundles, the delivery rates of bulk bundles will necessarily decrease. Due to the brevity of this test run compared to the large and medium queue tests, we effectively have a snapshot where the forwarding strategy is working but not enough high priority bundle deliveries exist to make it apparent.

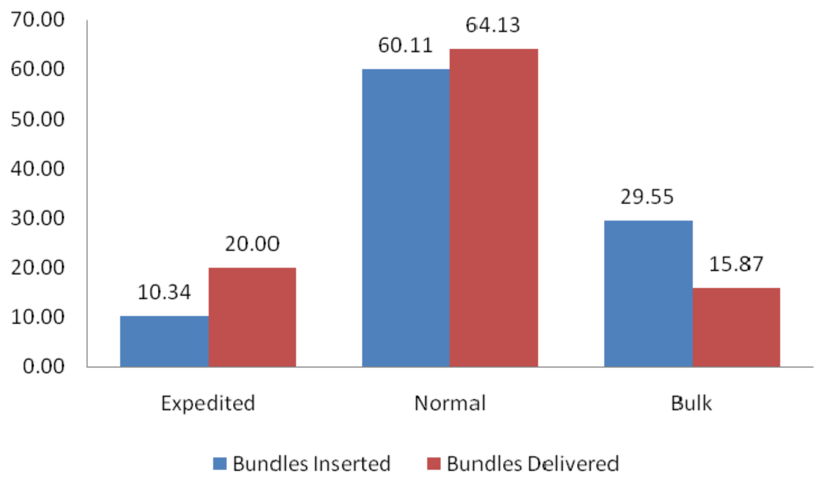

Figure 20: Insertion and Delivery Percentages for 10GB Queue Test Run Using COS and Priority

In Figure 20, we see a visual representation of what the numbers in Table 4 suggested. Here we can clearly see that the COS forwarding strategy has had a positive effect on the delivery percentages of the expedited and normal priority bundles; while also decreasing the number of bulk bundles making it through the system. In Figure 21 we can see that this observation holds for the medium queue test also.

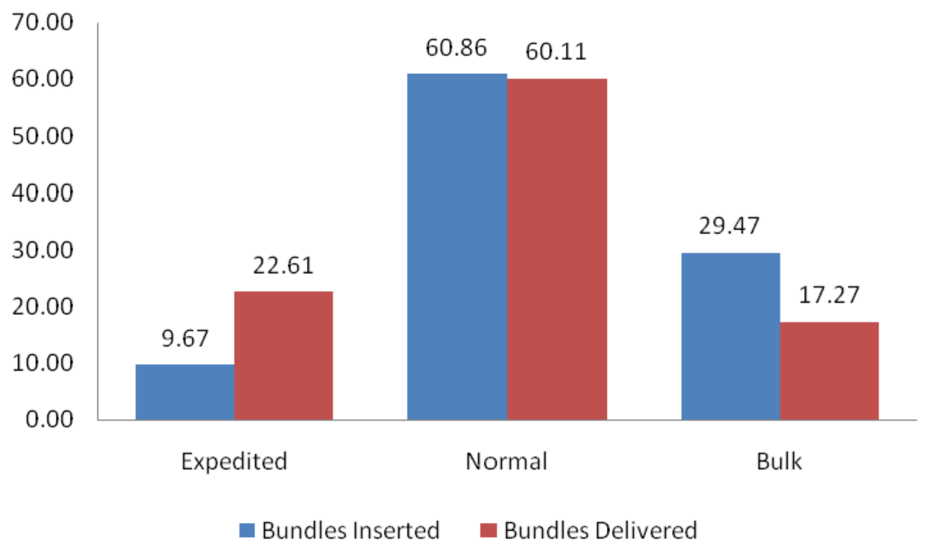

Figure 21: Insertion and Delivery Percentages for the 1GB Queue Test Run Using COS and Priority

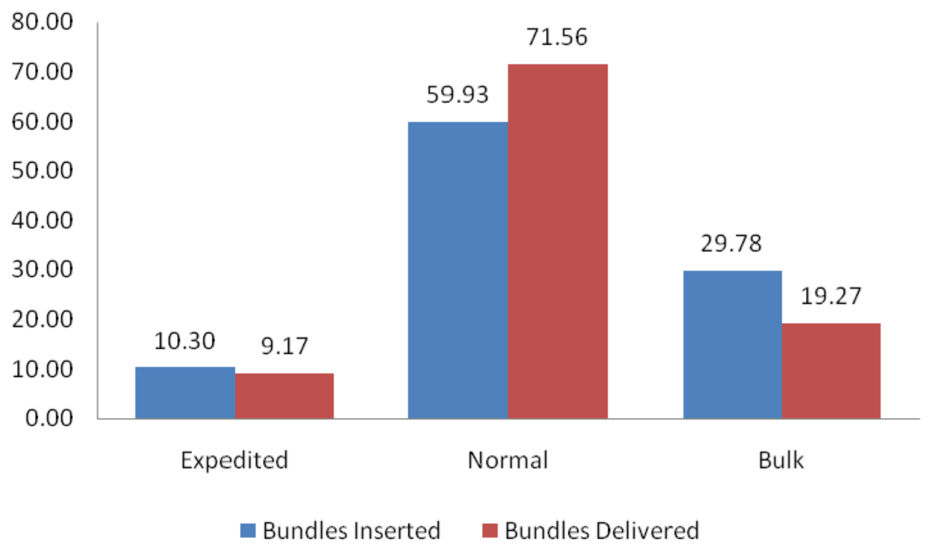

Figure 22: Insertion and Delivery Percentages for the 100MB Queue Test Run Using COS and Priority

In Figure 21, we see the evidence of the data shift due to the brevity of the test. The expedited bundles are delivered at a rate approximately equal to their insertion because there are so few of them present in the network. We can still see the effects of the COS forwarding strategy though. The more common normal priority bundles show a large increase in their delivery, while the bulk bundles still show a decrease.

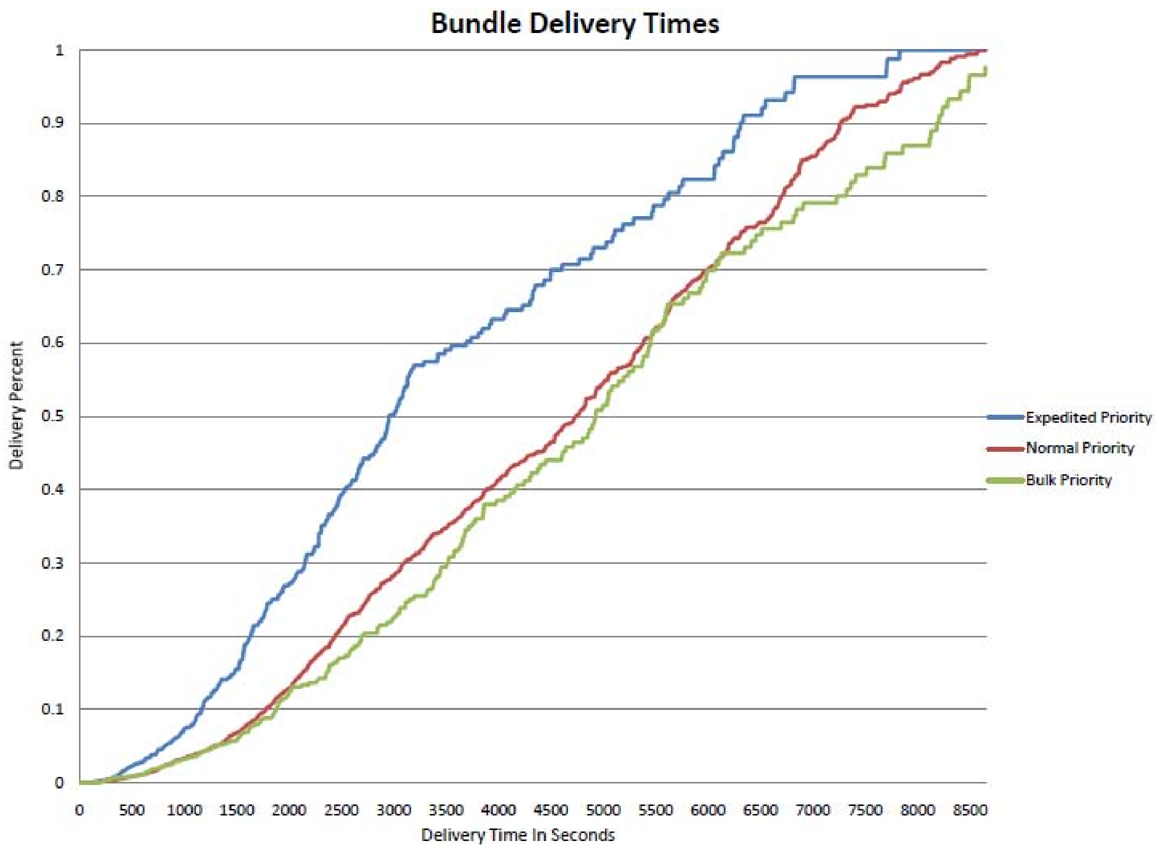

Figure 23: CDF of Delivery Times for the 10GB Queue Test Run Using COS and Priority

In Figure 23 and Figure 24, we can clearly see the effects our forwarding strategy had on the delivery times for both expedited and normal bundles. It is reasonable to infer that if the test could be run for an extended period of time, the divergence of the three priorities would be even more pronounced.

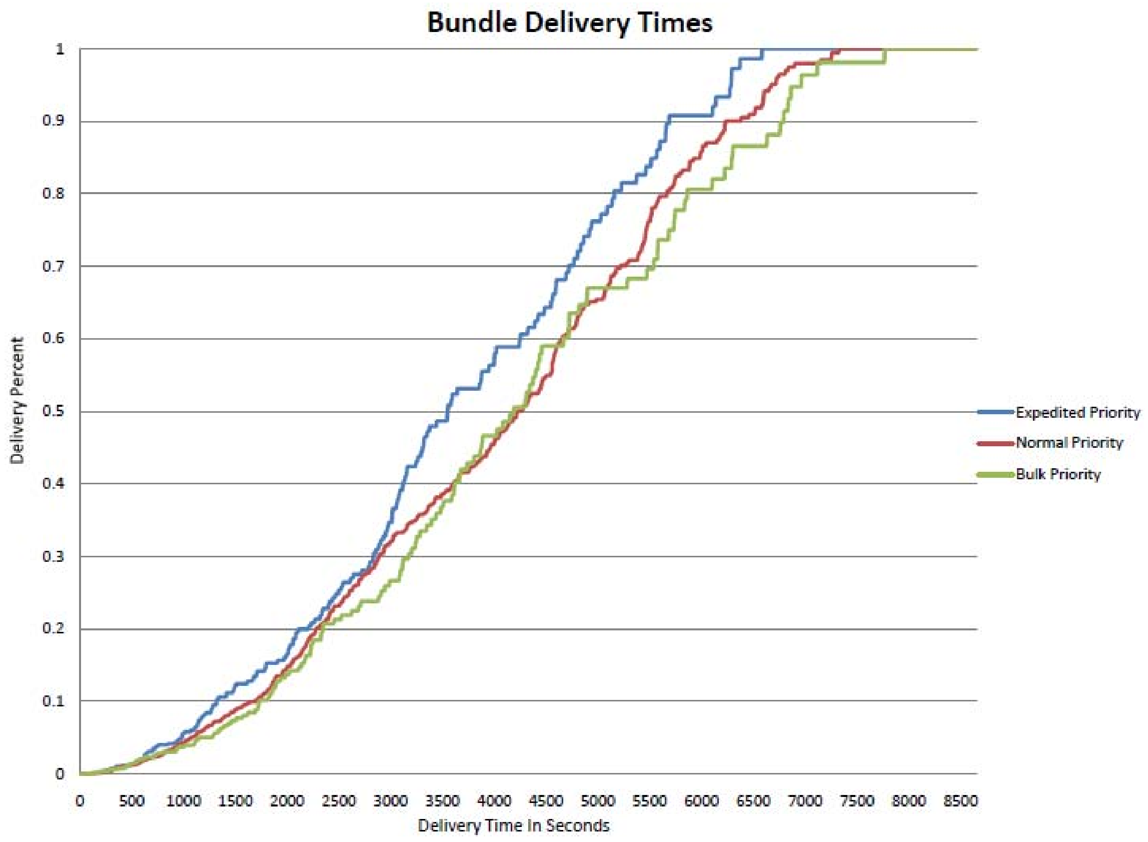

Figure 24: CDF of Delivery Times for 1GB Queue Test Run Using COS and Priority

In Figure 25, we can see that even within an extremely small dataset the COS forwarding strategy has broken out the three bundle types into three distinct lines. The expedited bundles arrive first, in the shortest amount of time, followed by the normal bundles, and finally the bulk bundles. These results are the proof of concept for the COS forwarding strategy. Even inside very small datasets, we can see the appropriate functioning of the strategy, and reduced delivery times for higher priority bundles.

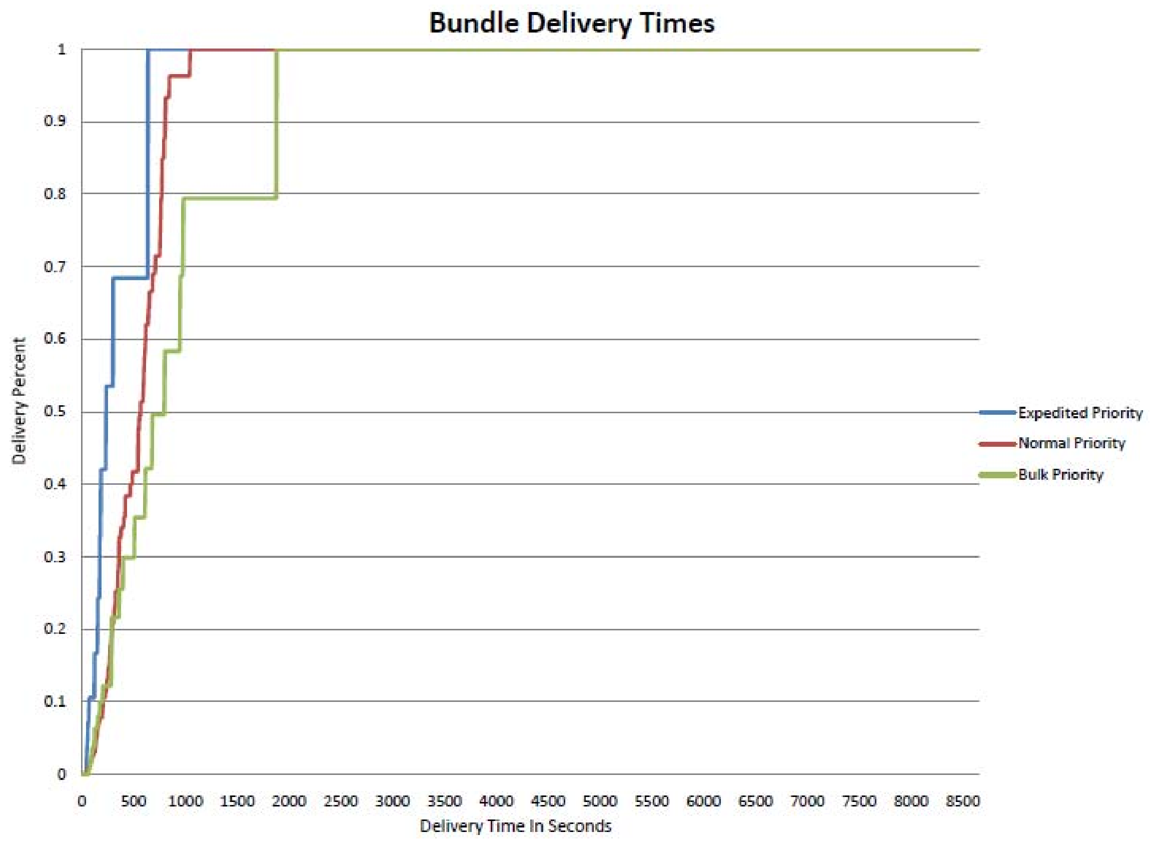

Figure 25: CDF of Delivery Times for the 100MB Queue Test Run Using COS and Priority

b. *COSFLEX Forwarding Strategy*

In the COSFLEX forwarding strategy, we attempt to control queue size by limiting the number of times a bundle can be forwarded based upon its priority and the size of the network it is traversing. Unfortunately, the node stability issue prevented us from being able to effectively test this functionality. With the nodes failing before the bundles had sufficient time to travel to places other than their destination, and the node failures reducing the size of the network. In addition, the random nature of the network does not take full advantage of PRoPHET’s ability to predict which node will be the best next hop for a bundle.

Even with node stability preventing the full expected effect of the COSFLEX strategy, it is worthwhile to perform some testing on it to see if it outperforms the COS strategy and/or still offers improvement over the baseline testing.

COS+ run large Queue	A	B	C	D	E	F	G	H
Life DD:HH:MM	1:00:49	3:08:48	1:00:30	1:00:18	1:00:02	1:00:18	1:00:04	1:00:35
High Injected	67	272	76	78	61	70	68	73
Med Injected	433	1473	446	444	429	441	439	452
Low Injected	251	705	219	225	222	229	219	223
High Delivered	63	28	16	31	18	14	32	3
Med Delivered	128	77	36	44	39	3	65	1
Low Delivered	35	16	10	15	1	2	4	0
High Del Time	2346	3088	2335	2199	3136	3936	2693	989
Med Del Time	2175	3130	1522	2701	4178	91	3538	276
Low Del Time	1359	3493	1527	1641	23	185	5045	0
Avg Queue Length	604	657	402	535	460	403	522	445

Total injected
765
4557
2293
Total Delivered
205
393
83
Avg Delivery Times (Sec)
2590.3
2201.4
1659.1
Avg Queue Size (bundles)
503.5

Table 7. 10GB Queue Test Run Using COSFLEX and Priority

In this first test run (Table 7), we see results similar to the COS test run. The difference here is, we see a pronounced increase the number of high priority bundles that were successfully delivered over the previous test runs. The node failures and the manner of their failing was the same as all previous testing runs. On the surface, it appears that this jump in expedited bundles being delivered is due to the forwarding strategy allowing expedited priority bundles to be forwarded with no restrictions. In this manner, they can effectively be flooded across the network within the restrictions of the GRTR forwarding strategy that is also used as a base test for the COSFLEX strategy.

COS+ run med Queue	A	B	C	D	E	F	G	H
Life DD:HH:MM	0:22:43	4:05:16	4:05:16	1:00:58	1:05:26	0:23:34	1:01:08	1:00:16
High Injected	68	296	294	68	85	83	77	68
Med Injected	404	1817	1823	457	501	436	438	425
Low Injected	211	959	878	212	283	208	209	230
High Delivered	41	70	61	17	17	12	19	0
Med Delivered	119	285	85	67	12	29	37	0
Low Delivered	30	64	5	15	1	1	2	0
High Del Time	2783	2918	3769	1617	1800	1866	3835	0
Med Del Time	2708	4934	5011	1787	2340	2818	4382	0
Low Del Time	2296	3712	3393	1894	167	29	5726	0
Avg Queue Length	594	657	654	546	479	431	526	436

Total injected
1039
6301
3190

Total Delivered
237
634
118

Avg Delivery Times (Sec)
2323.5
2997.5
2152.1

Avg Queue Size (bundles)
540.38

Table 8. 1GB Queue Test Run Using COSFLEX and Priority

In Table 8, we again see the large jump in expedited priority bundles that we saw during the first test and Table 7. This is not unexpected, as we have stated earlier, due to node failures occurring before the queues become full, the reduction of the queue size has little effect on the test and the results should have been similar to the first COSFLEX test run.

COS+ run small Queue	A	B	C	D	E	F	G	H	
Life DD:HH:MM	0:08:28	0:03:37	0:17:36	0:17:25	0:06:03	0:05:33	0:08:21	0:03:28	Total injected
High Injected	28	6	51	55	14	12	32	16	214
Med Injected	173	66	310	299	116	92	159	68	1283
Low Injected	81	23	158	158	52	50	87	30	639
									Total Delivered
High Delivered	1	0	3	2	1	2	4	3	16
Med Delivered	13	7	25	3	5	4	2	12	71
Low Delivered	11	1	16	18	4	4	0	3	57
									Avg Delivery Times (Sec)
High Del Time	984	0	445	422	171	32	719	258	378.88
Med Del Time	564	362	454	413	311	272	1384	294	506.75
Low Del Time	556	48	914	1094	377	266	0	408	457.88
									Avg Queue Size (bundles)
Avg Queue Length	92	71	292	92	78	68	91	94	109.75

Table 9. 100MB Queue Test Run Using COSFLEX and Priority

In Table 9, we observe some strange behavior. In other small queue runs we mention the abnormally large number of bulk and normal priority bundles that get delivered. We posited earlier that this was due to fewer expedited bundles being available for delivery, and therefore more of the lower priority bundles are delivered. In this test, we see a very large number of bulk bundles delivered. This is likely due to the same reason stated before; however, this is the first time we have seen a difference this pronounced. It is not clear from this single data point if this is an artifact of the COSFLEX forwarding strategy, or just an effect of our random network. Further testing with more stable network nodes should help clarify this point.

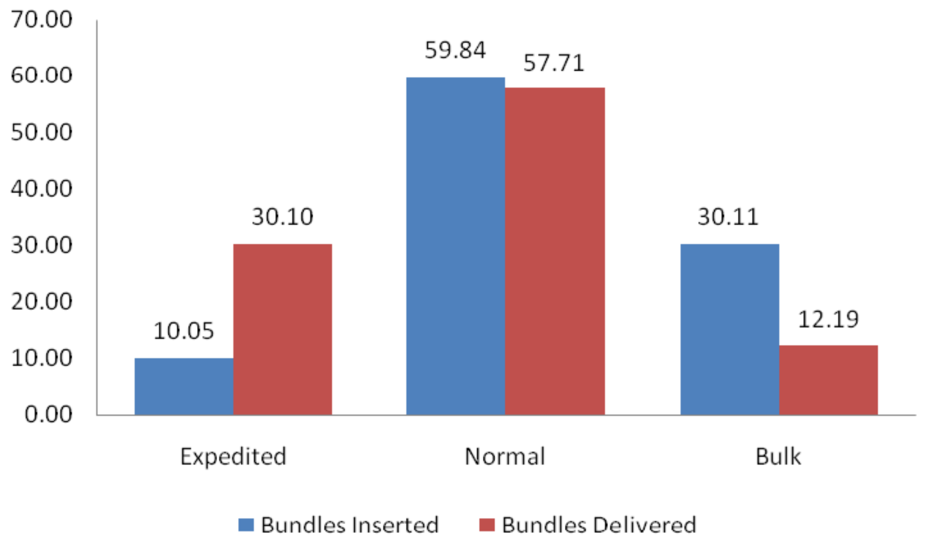

Figure 26: Insertion and Delivery Percentages for the 10GB Queue Test Run Using COSFLEX and Priority

In Figures 26 and 27, we can see the marked improvement in the percentage of higher priority bundles that are delivered to their destinations. It appears that the COSFLEX forwarding strategy has provided another level of improvement for higher priority bundles. In Figure 28 we can see a comparison of the delivery percentages for the three 10GB queue tests. In it we can see the performance advantage for expedited bundles provided by our new forwarding strategies, little a small effect on the normal priority bundles, and a sharp decrease in bulk bundles. We saw 20 and 22 percent of the bundles delivered had high priority in the large and medium COS tests. Here we see 30 and 24 percent, this improvement and resulting decrease in low priority bundles make sense with the added restrictions placed on forwarding in the COSFLEX strategy.

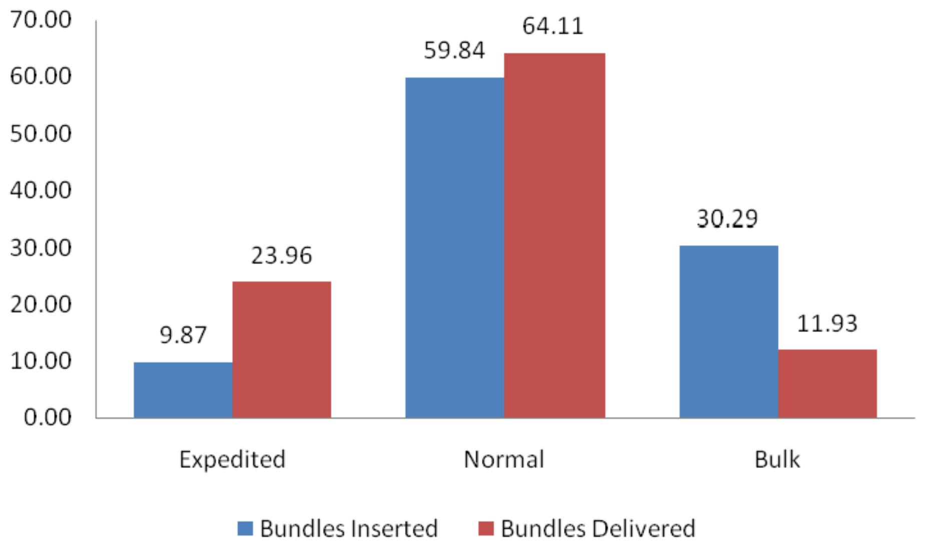

Figure 27: Insertion and Delivery Percentages for the 1GB Queue Test Run Using COSFLEX and Priority

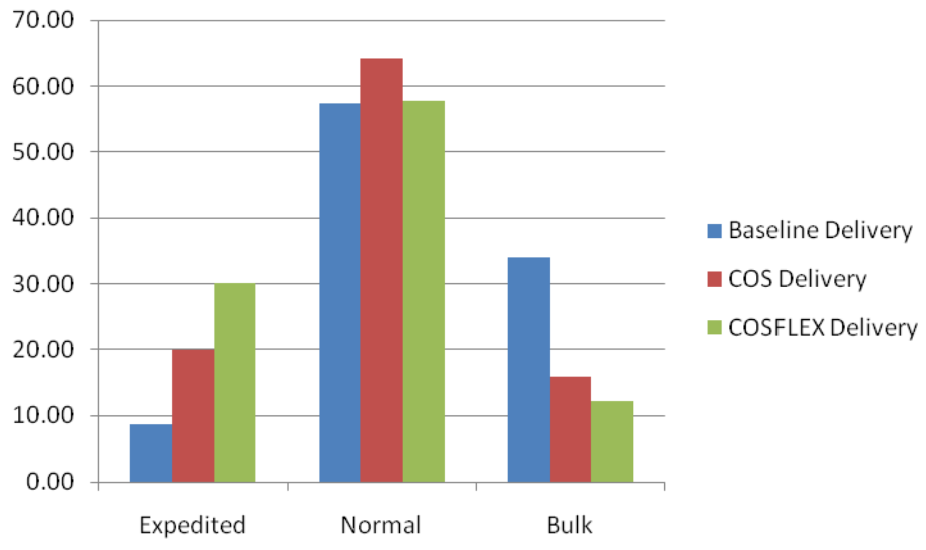

Figure 28: Comparison of Delivery Percentages for Baseline, COS, and COSFLEX Tests With 10GB Queues

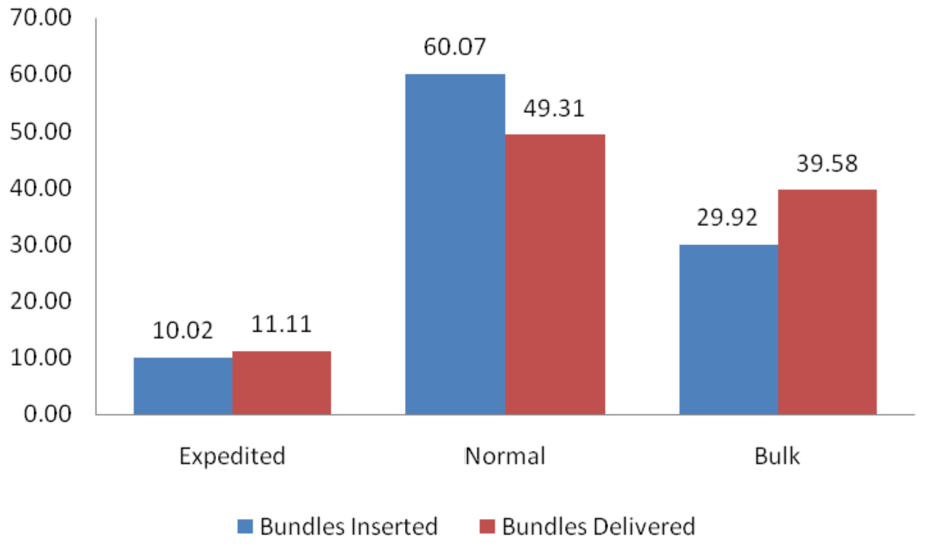

Figure 27: Insertion and Delivery Percentages for the 100MB Queue Test Run Using COSFLEX and Priority

Figure 29 clearly shows the strange performance shift observed in our short small queue test run. In this run, it appears that exactly the opposite of the intended network performance is being observed. It seems as if the bulk bundles are being forwarded at the expense of the expedited and normal priority bundles. We believe that this is an artifact of the shortened run, and that if it were run successfully for a longer period of time, we would see results similar to those in Figures 26 and 27.

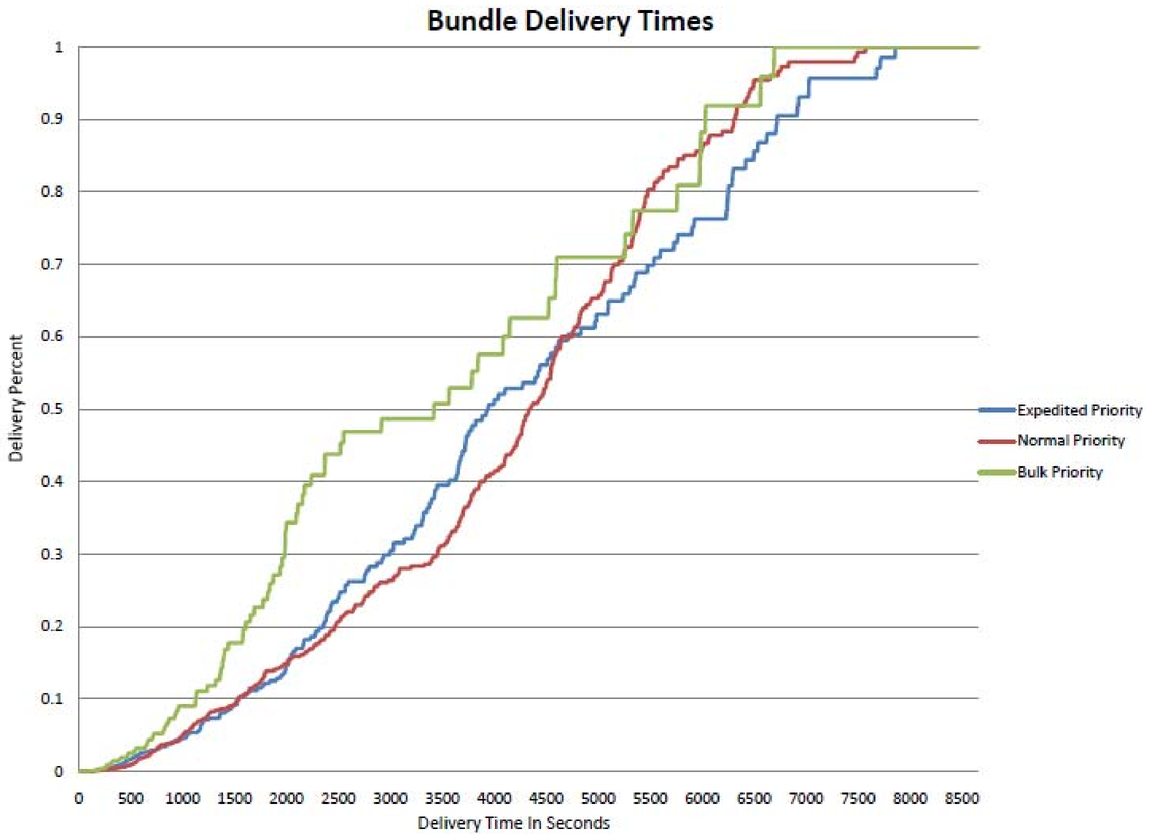

Figure 28: CDF of Delivery Times for the 10GB Queue Test Run Using COSFLEX and Priority

Figure 30 shows initially counterintuitive results. During the COS test runs, we could clearly see an improvement of delivery times for the expedited bundles at the expense of the lower priority bundles. In this test the opposite is true. When this graph is looked at outside of any context, it appears that not only has COSFLEX failed to achieve the desired results but accomplished exactly the opposite. Taken in context of the results seen in our 100MB queue runs, we form the following hypothesis: bulk bundles only get forwarded on a regular basis at the beginning of a test run because of the short connection times and large amounts of data in the queue as the run progresses. For this reason, the majority of bulk bundles only get delivered at the beginning of the run, and thusly, have very small delivery times. This can be seen by carefully examining the charts and noting that line for the bulk bundles is smoother, less step like, in the regions below the 2000-2500 second delivery time. After this, it becomes ragged as the bundles delivered become fewer and farther in between. During a longer test run, we believe that the expedited and normal bundles would eventually surpass the bulk bundles as more of them are delivered at shorter times, and the bulk bundles that eventually make it to their destinations have longer and longer delivery delays.

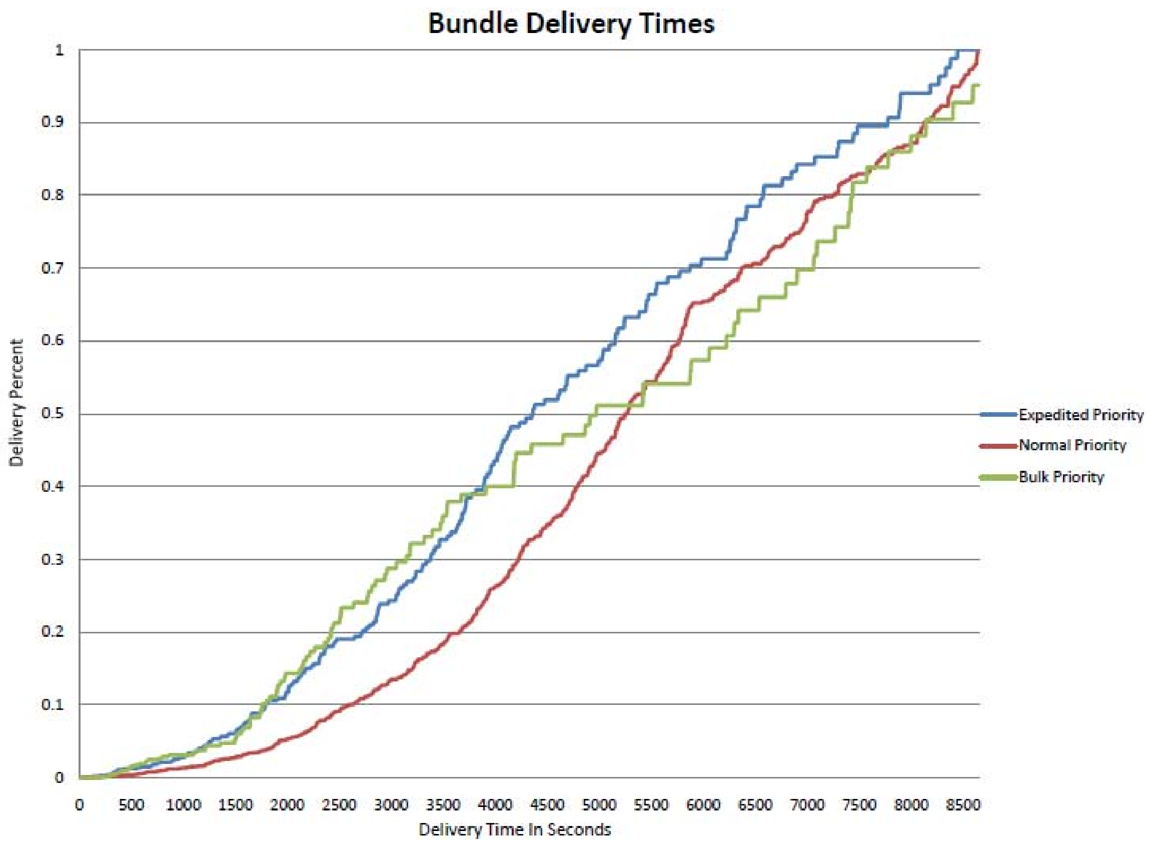

Figure 29: CDF of Delivery Times for the 1GB Queue Test Run Using COSFLEX and Priority

Figure 31 shows the same initial jump by the bulk bundles we see in Figure 30, except here we see the expedited and normal bundles eventually climbing above the bulk line as delivery time increases. This appears to support the hypothesis we proposed for Figure 30. We believe strongly that longer test runs would validate the expectations for this forwarding policy in the face of this seemingly contrary data.

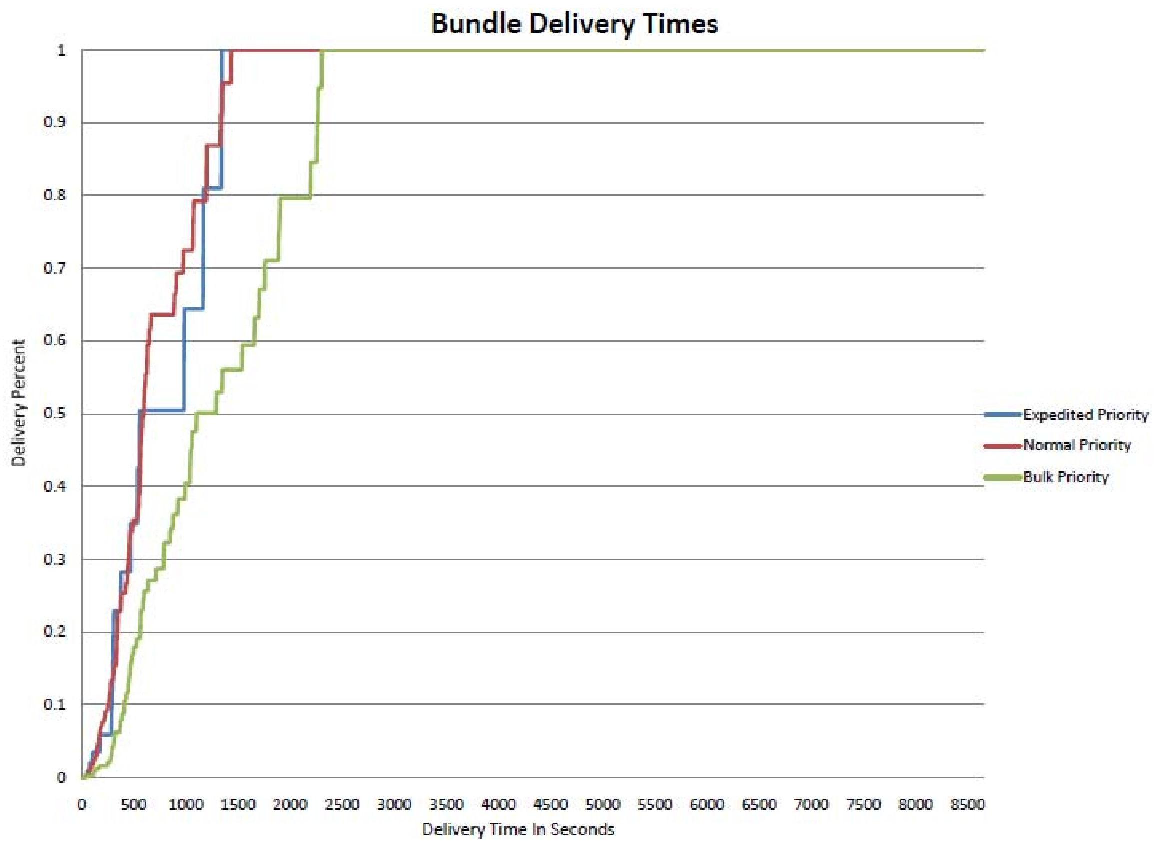

Figure 30: CDF of Delivery Times for the 100MB Queue Test Run Using COSFLEX and Priority

In Figure 32, we see results similar to the previous tests runs with the smallest queue size. It is interesting to note, however, that this is the only test run where the delivery times seem to be what we would have expected, even with the number of bundles delivered for each priority being exactly the opposite.

The performance of the COSFLEX forwarding strategy shows promise to improve even more the amount of higher priority data that gets through the network. Due to the conflicting results seen, high delivery rates, and long delivery times, further testing is warranted to examine exactly why these results are presenting themselves this way.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND FUTURE WORK

The primary motivation of this thesis was to fill a need for the development and evaluation of new, priority based, message-forwarding strategies based on the existing DTN2 reference code base. Additionally, we sought to evaluate the performance of this modified DTN routing scheme on a live test-bed. We decided to focus development on the code of the PRoPHET router module. This was due to the ability to quickly develop our routing code in PRoPHET, and PRoPHET's ability to make decisions based on the properties of the bundles contained at a node and what nodes it has connected to in the past.

A. THESIS SUMMARY

The development of the code was relatively simple; however, testing revealed several bugs in the original distributed code of the PRoPHET router that severely delayed performance testing of the newly developed routing algorithms. Every effort was made to remove the bugs we found, and mitigate those that we could not. This resulted in a more stable router, but node stability problems were still an issue in the data gathering phase of the experiment.

With our limited timeframe, we still managed a proof of concept confirmation that using a priority sensitive routing system can be implemented inside of a DTN to provide effective service differentiation based on bundle priority. As the data shows, adding a priority aware forwarding strategy allows the router to expedite the traversal of the network by certain bundles over others. However, our data set is extremely limited due to the amount of time spent attempting to stabilize the nodes. Reviewing the questions posed earlier in this thesis, we will assess our success and failings in answering each.

The first question was, “Given a real DTN topology what are the baseline performance metrics under varying network loads?” We managed to take a look at this question by examining the standard PRoPHET router with a decreasing queue size using the GRTR forwarding strategy and the FIFO queuing policy. However, due to the stability issues we encountered during the test runs; our data set is very small and only provides us with a limited glimpse at the performance of our network. This was enough, however, to provide us with a comparison data set for our new forwarding strategies.

The second question, “At what point does message overload reduce the network performance to unacceptable levels?” was not answered in any way. With the nodes in the test bed failing when either a bundle expired or the queue filled up, we were unable to observe a network under stress by large numbers of data bundles. As such, we cannot answer this question and are forced to leave it to future researchers experimenting with a more stable platform.

The third question posed, “Is it possible to accommodate prioritization in a DTN routing protocol on a DTN topology to allow QoS metrics to be determined?” is a definite “yes.” By adding priority to the PRoPHET router, we showed that it could be used as a filter for bundles in the network. While using the COS strategy, we observed a gain of 10% over the baseline expedited delivery rate, and using COSFLEX, observed a 20% increase over baseline for expedited bundles. Given a stable routing platform that implements both forwarding and queuing policies, it is possible to determine metrics that could be used to differentiate quality of service. Given our limited data set we will not make any claims here. One data point does not allow for effective comparison, but the proof of concept suggests that COS and COSFLEX are promising approaches in DTN routing. Further testing resulting in a suitable dataset will allow the improvements to be quantified for specific network architectures.

Fourth we wondered, “What are the appropriate comparative QoS metrics seen in our topology?” We saw that the delivery rates for the bundles of differing priorities and delivery times were good comparative metrics. As in the previous question, we cannot make a prediction regarding the comparison between them based on only one data point for each queue size. Further testing on an improved test bed or more stable system would certainly allow this to be done.

Fifth, we considered, “Does the highest QoS category maintain acceptable performance even under conditions that made performance unacceptable in the unmodified network?” This question is also unanswered due to the fact we were unable to put any stress on the nodes in the network without the nodes failing. This is still a very interesting question to examine in the future.

Finally, “What new security vulnerabilities does this modified network routing introduce into delay tolerant networks?” During the development of code, we identified three areas of weaknesses associated with the addition of priority based forwarding into the DTN router.

1. Badly behaved or naive users could tag all their bundles with expedited priority fields attempting to ensure better service.
2. Currently, the COS and COSFLEX routing algorithms will not forward a bundle if it has an undefined priority field. If an attacker could modify or corrupt this field, it could prevent high priority data in the network from being delivered.
3. Flooding the network with expedited bundles creates an effective denial of service for lower classes, and a near denial of service for expedited traffic. This attack would immediately flush normal and bulk priority traffic out of the network queues and soon get rid of expedited traffic. This is due to the queuing policy dropping the oldest, lowest priority bundle first. This results in the eviction of legitimate bulk and normal priority bundles, and eventually expedited data as a nodes queue reaches capacity. If the attacker continues flooding the node with bogus high priority traffic eventually all the legitimate traffic will be removed.

   This attack could also be affected by inserting an expedited priority bundle at each node in the network that is too large to be transmitted during any contact interval. Not only will this delete valid data in the queue, but prevent the node from sending data until the bundle is deleted by either timing out or having enough valid expedited bundles arrive to make it the candidate for eviction from the queue.

In addition to the questions posed in the introduction, there is continuing work on the Internet Draft defining the PRoPHET router system. However, the version of the router module distributed with the DTN2 system has not been updated since 2007. During the 76th meeting of the IETF in Hiroshima, Japan, Kevin Fall presented a PowerPoint presentation titled, “DTN Reference Implementation Update” [19]. On page eight of this presentation, the following questions were posed about PRoPHET:

- Why is it implemented in a different way to other routers?
- Is anyone using it?
- Is it still an accurate implementation of the draft?
- Does it work?
- Can it be removed/altered?

While this thesis is not aimed to directly answer these questions, it lends some insight to the last two.

The PRoPHET router as distributed with the DTN2 software from sourceforge does work; however, it appears that there are several problems with it. These issues allow it to remain a representation of the Internet draft [3], but not an accurate one. A major part of the PRoPHET router is its ability to manage the queue in relation to some criterion. Currently this does not happen. PRoPHET defers to the DTN daemon and simply refuses to accept any more bundles until space in the queue is created. This results in the queuing strategy being completely bypassed. When the PRoPHET router receives a bundle and attempts to add it to its list, it first asks the DTN daemon if it should. If the DTN daemon returns false, then PRoPHET exits the handle_bundle_received() routine and never adds the bundle or attempts to call evict() to make room for it. With this operation, only the forwarding strategy is implemented. Once this block to using the queuing strategy is removed, the node becomes unstable and has issues with segmentation faults. The evict() function of the PRoPHET router was changed to help alleviate these issues, but no iteration ever successfully prevented the node from crashing when evict() was called. In addition, issues were encountered when a bundle expired. No time remained to hunt down the expiration issues so our experiments and data set were limited. The nodes in our test runs all lived about an hour and encountered segmentation faults when the PRoPHET handle_bundle_expired() routine is called. This was confirmed by the stack trace showing a call from the method to PRoPHET's find function as the culprit.

In response to the second question posed by Mr. Fall, "can PRoPHET be removed/altered," the answer is yes on both counts. PRoPHET, as it is currently distributed has flaws that prevent it from operating as the internet draft [3] intends. In our opinion, PRoPHET should fix these deficiencies and can then continue to be distributed, remove it from the distribution until a fixed and edge case testing proves that the underlying bugs are resolved.

In summary, we managed to show that COS forwarding is a useful technique for bundle routing inside a DTN. The COSFLEX strategy also shows promise, but further testing on both is needed before any comparative metrics can be discerned. This functionality should be added to the DTN daemon, and distributed to allow a wider spectrum of uses in the future. PRoPHET, while an interesting prospect, needs some reworking and edge case testing before it can be fully realized in the spirit of the Internet Draft.

B. FUTURE WORK

The following areas merit future work and can be good thesis research topics.

- Stabilization of the PRoPHET router. This was the major impediment to the work in this thesis. The code itself has not been updated in the four years prior to this thesis, and the DTN2 code it runs with has been undergoing continual development. Can the PRoPHET code be brought back up to the level of the DTN code, and if not, can a new version be created to fulfill the same purpose without the bugs?

- Development of a similar router in the DTN code. PRoPHET was selected due to the simplicity of expanding its code to allow forwarding based on priority criteria. With the instability inherent in PRoPHET, could a similar router be developed and implemented that is more stable?

- Repeating experiments on a larger network and varying the PRoPHET settings. The work to stabilize the router itself robbed us of much of the time we had hoped to spend varying our testing parameters in order to develop a richer data set. Due to this, our data provides a proof of concept result, but further testing on a larger test bed and varying the testing parameters will further develop and refine the performance characteristics of the routing system.

- Further investigation of the COSFLEX routing system. The data we achieved from the test runs using COSFLEX seems counterintuitive. If we saw high delivery percentages, it was paired with slow delivery times, and vice versa. Was this a function of our test bed, or some kind of artifact of the way forwarding is done in this system?
- Testing of the code developed with varying mobility models. PRoPHET takes advantage of the history of encounters nodes have made. This makes it ideal for nonrandom mobility models. Testing in this thesis only used a random model, would results be improved or degraded in a nonrandom one?
- Creation of the ability for DTN to modify sorting by the priority value. Currently in the DTN code this merely contains the comment "needs to be implemented." If this could be implemented so that the base DTN stores bundles in a list sorted by priority, then PRoPHET would no longer be needed, and the DTN routers already implemented could be used for further testing.

- Adaptation of the DTN daemon and its associated routers to wireless phones. The current implementation of DTN is written in the C and C++ languages. These languages are not supported by many of the popular smart phones today. The adaptation of these to the Android or iPhone platforms would greatly expand the research areas available for DTN topologies.
- Development of a system to classify and automatically assign priority to bundles. In this thesis we assumed trusted and well behaved users. This is rarely the case in a real environment. Can policies be implemented to allow the system to automatically assign priorities to bundles as they are created without user input?

C. RECOMMENDATIONS

Anyone doing further development or testing work with DTN technology would be well advised to become a member of the DTNRG mailing lists. Having become a member of both the “users” and “info” mailing lists, the assistance that the members of these lists provided during this thesis has been invaluable. The members of these lists are active researchers in the field of DTN and can provide insight and advice when any future developers hit a wall.

In addition the book *Delay and Disruption Tolerant Networking* by Stephen Farrell and Vinny Cahill was a great help. It is available on Amazon.com but is fairly expensive. It was well worth the money in our opinion and any future researchers should use it as a background work to quickly develop a useful knowledge base about DTNs their use.

APPENDIX

A. PROPHET.SH

```
#start DTN daemon script with prophet

#!/bin/bash

datestring=$(date +%Y-%m-%d-%H:%M)

newlog="dtnd-h-$datestring.log"

cd ~/Desktop/dtn2/dtn-2.7.0/daemon
dtnd -c ~/Desktop/dtn/dtn_prophet.conf -o ~/Desktop/dtn/logs/$newlog -l
info -s 584326 –t
```

B. RECEIVE.SH

```
#Starts the service at a DTN node that consumes bundles when #delivered
#test bundle reception
#!/bin/bash

cd /usr/dtn2/dtn-2.7.0/apps
      dtnrecv dtn://node_h.dtn/test
```

C. BUNDLE_SEND.SH

```
#test bundle sender
#!/bin/bash
cd ~/Desktop/dtn2/dtn-2.7.0/apps

#---------------------------------------------------------------
#                 Variables Section
#---------------------------------------------------------------

modulo=100 #Modulus to limit random mumbers to between 0 and 99
sendSwitch=0 #adjustable priority to determine if bundle is sent
prob=49 #probability bundle is sent on each iteration of the loop
      #49 = a 50/50 chance of sending or sitting idle.
destination="" #destination variable for the bundle
destSwitch=0 #case variable for destination selection
priority="" #Priority variable for the bundle
priSwitch=0 #case variable for priority selection

while [ 1 ]; do #loop until disrupted with Ctrl-C

#---------------------------------------------------------------
#           Determine if a bundle should be sent
#---------------------------------------------------------------
```

```
      sendSwitch=$RANDOM
      let "sendSwitch %= $modulo"

      if [ $sendSwitch -le $prob ]; then

#----------------------------------------------------------------
#           Determine Bundle Destination
#----------------------------------------------------------------

      destSwitch=$RANDOM
      let "destSwitch %= 7"

      case "$destSwitch" in
                  [0]  )      destination="dtn://node_a.dtn/test";;
                  [1]  )      destination="dtn://node_b.dtn/test";;
                  [2]  )      destination="dtn://node_c.dtn/test";;
                  [3]  )      destination="dtn://node_d.dtn/test";;
                  [4]  )      destination="dtn://node_e.dtn/test";;
                  [5]  )      destination="dtn://node_f.dtn/test";;
                  [6]  )      destination="dtn://node_g.dtn/test";;
      #           [6]  )      destination="dtn://node_h.dtn/test";;
      esac

      #echo "Destination:  $destination"

#----------------------------------------------------------------
#                 Determine Bundle Priority
#----------------------------------------------------------------

      priSwitch=$RANDOM
      let "priSwitch %= $modulo" #Generate priority

      case "$priSwitch" in
            [0-9]             )     priority='expedited';;
            [1-6][0-9]        )     priority='normal';;
            [7-9][0-9]        )     priority='bulk';;
      esac

      #echo "Priority:  $priority"

#----------------------------------------------------------------
#                 Craft DTNSEND Command line
#----------------------------------------------------------------

      #10 MB file is named : test10MB.img
      #100 MB file is named :test100MB.img
      #1 MB file is named :  test1MB.img

      dtnsend -P $priority -e 8640 -W -s dtn://node_h.dtn –d
             $destination -t f -p ~/Desktop/test1MB.img
      echo Bundle sent with $priority priority to $destination.
      fi
sleep 6
done
```

D. DISRUPT.SH

```
#network disruption generator
#NOTE:  MUST BE RUN AS ROOT!!


#!/bin/bash

#----------------------------------------------------------------
#                 Variables Section
#----------------------------------------------------------------
upMod=60    #used to select the number of minutes plus 1
            #the node will be up resulting in an uptime
            #between 1 and 2 minutes

uptime=0    #the number of seconds the node will be up

downMod=180       #used to select the number of minutes plus 3
            #the node will be up resulting in a downtime
            #between 3 and 4.5 minutes

downtime=0  #number of seconds the node will be down

#randomly sleep for 1-6 minutes to start off.
#nodes start off with networking disabled.

      downtime=$RANDOM
      let "downtime %= 300"
      let "downtime += 60"
      sleep $downtime

#create disruptions until cancelled

while [ 1 ]; do

#network up for between 1 and 2 minutes

      uptime=$RANDOM
      let "uptime %= $upMod"
      let "uptime += 60"
      ifconfig eth0 up
      sleep $uptime

#network down for between 3 and 4.5 mintues

      downtime=$RANDOM
      let "downtime %= $downMod"
      let "downtime += 90"
      ifconfig eth0 down
      sleep $downtime
done
```

E. SCRAPER.PY

```
#======================================================
#Python script for gathering data from DTN log files.
#Created by LCDR Christopher Rapin February 15 2011.
#======================================================

#     Variables

hipri = 0
medpri = 0
lowpri = 0

hidel = 0
meddel = 0
lowdel = 0

hitime = 0
medtime = 0
lowtime = 0

queueupdate = 0
queuetotal = 0

start = 0
end = 0

queuelenTOTAL = 0
queuelenAVG = 0

#     Input and output files

outfile = '/home/dtn/Desktop/Base_1/data_g.log'
infile = "/home/dtn/Desktop/Base_1/dtnd-g-2011-02-25-16_01.log"

log = open(infile,'r')

parse = open(outfile, 'w')

#     Read in lines from the file

linelist = log.readlines()
log.close()

#     Grab times from first and last lines for age calculation

start = float(linelist[0][1:18])
end = float(linelist[len(linelist)-1][1:18])

#     Parse the list of lines to gather data

for line in linelist:
      if (line.find("RAPIN") >= 0):
            if (line.find("CREATION") >= 0):
                  if (line.find("Priority: 0") >= 0):
                        lowpri += 1
                  elif (line.find("Priority: 1") >= 0):
                        medpri += 1
                  elif (line.find("Priority: 2") >= 0):
                        hipri += 1
            elif (line.find("DELIVERED") >= 0):
                  if (line.find("Priority: 0") >= 0):
                        lowdel += 1
                        lowtime += float(line[100:(len(line)-1)])
                  elif (line.find("Priority: 1") >= 0):
                        meddel += 1
                        medtime += float(line[100:(len(line)-1)])
                  elif (line.find("Priority: 2") >= 0):
                        hidel += 1
                        hitime += float(line[100:(len(line)-1)])
            elif (line.find("Queue") >= 0):
                  queueupdate += 1
                  queuetotal += int(line[73:])


#     Output data to the output file.

string = "Number of high priority bundles injected: ", hipri
parse.write(str(string))
parse.write('\n')

string = "Number of medium priority bundles injected: ", medpri
parse.write(str(string))
parse.write('\n')

string = "Number of low priority bundles injected: ", lowpri
parse.write(str(string))
parse.write('\n')
parse.write('\n')

string = "% of high priority bundles injected: ",
((float(hipri)/(hipri+medpri+lowpri))*100)
parse.write(str(string))
parse.write('\n')

string = "% of med priority bundles injected: ",
((float(medpri)/(hipri+medpri+lowpri))*100)
parse.write(str(string))
parse.write('\n')

string = "% of low priority bundles injected: ",
((float(lowpri)/(hipri+medpri+lowpri))*100)
parse.write(str(string))
parse.write('\n')
parse.write('\n')

string = "Number of high priority bundles delivered: ", hidel
parse.write(str(string))
parse.write('\n')

string = "Number of medium priority bundles delivered: ", meddel
```

```
parse.write(str(string))
parse.write('\n')

string = "Number of low priority bundles delivered: ", lowdel
parse.write(str(string))
parse.write('\n')
parse.write('\n')

if (hidel != 0):
      string = "Average Delivery time for high priority bundle: ",
hitime/hidel
else:
      string = "Average Delivery time for high priority bundle: 0"
parse.write(str(string))
parse.write('\n')

if (meddel != 0):
      string = "Average Delivery time for medium priority bundle: ",
float(medtime)/meddel
else:
      string = "Average Delivery time for medium priority bundle: 0"
parse.write(str(string))
parse.write('\n')

if (lowdel != 0):
      string = "Average Delivery time for low priority bundle: ",
float(lowtime)/lowdel
else:
      string = "Average Delivery time for low priority bundle: 0"
parse.write(str(string))
parse.write('\n')
parse.write('\n')

string = "Average queue length during the run: ",
float(queuetotal)/queueupdate
parse.write(str(string))
parse.write('\n')
parse.write('\n')

parse.write("The node was running for :\n")
runtime = end - start
string = int(runtime)/8640, 'days'
parse.write(str(string))
parse.write('\n')
runtime = runtime % 8640

string = int(runtime)/360, "hours"
parse.write(str(string))
parse.write('\n')
runtime = runtime % 360

string = int(runtime)/6, "minutes"
parse.write(str(string))
parse.write('\n')
```

F.   CDF_SCRAPER.PY

```
#=======================================================#
#Python script for gathering data from DTN log files.  #
#Crawls all log files at once and gathers all delivery #
#times in three priority specific lists, output to a   #
#file when completed.                                   #
#Created by LCDR Christopher Rapin March 10 2011.      #
#=======================================================#


      hitime = []
      medtime = []
      lowtime = []


      outfile = '/home/carapin/Desktop/Base_1/CDF_Data.log'


      infile_a    =    "/home/carapin/Desktop/Base_1/dtnd-a-2011-02-25-
16_00.log"
      infile_b    =    "/home/carapin/Desktop/Base_1/dtnd-b-2011-02-25-
16_00.log"
      infile_c    =    "/home/carapin/Desktop/Base_1/dtnd-c-2011-02-25-
16_01.log"
      infile_d    =    "/home/carapin/Desktop/Base_1/dtnd-d-2011-02-25-
16_02.log"
      infile_e    =    "/home/carapin/Desktop/Base_1/dtnd-e-2011-02-25-
16_01.log"
      infile_f    =    "/home/carapin/Desktop/Base_1/dtnd-f-2011-02-25-
16_01.log"
      infile_g    =    "/home/carapin/Desktop/Base_1/dtnd-g-2011-02-25-
16_01.log"
      infile_h    =    "/home/carapin/Desktop/Base_1/dtnd-h-2011-02-25-
16_02.log"


      loglist                                                         =
[infile_a,infile_b,infile_c,infile_d,infile_e,infile_f,infile_g,infile_h
]
      linelist = []
      for x in loglist:
            log = open(x,'r')
            linelist = log.readlines()
            for line in linelist:
                  if (line.find("RAPIN") >= 0):
                        if (line.find("DELIVERED") >= 0):
```

```
                        if (line.find("Priority: 0") >= 0):

lowtime.append(float(line[100:(len(line)-1)]))
                        elif (line.find("Priority: 1") >= 0):

medtime.append(float(line[100:(len(line)-1)]))
                        elif (line.find("Priority: 2") >= 0):

hitime.append(float(line[100:(len(line)-1)]))
      log.close()

parse = open(outfile, 'w')

parse.write("High Priority times:")
parse.write('\n')
for value in hitime:
      parse.write(str(value))
      parse.write('\n')
parse.write('\n')

parse.write("Medium Priority times:")
parse.write('\n')
for value in medtime:
      parse.write(str(value))
      parse.write('\n')
parse.write('\n')

parse.write("Low Priority times:")
parse.write('\n')
for value in lowtime:
      parse.write(str(value))
      parse.write('\n')
parse.write('\n')
```

G. FWDSTRATEGY.H

```
--- orig_FwdStrategy.h  2010-02-21 08:11:18.000000000 -0800
+++ FwdStrategy.h 2011-02-03 09:01:50.000000000 -0800
@@ -40,7 +40,9 @@
         GRTR_PLUS,
         GTMX_PLUS,
         GRTR_SORT,
-        GRTR_MAX
+        GRTR_MAX,
+     COS,  //Inserted for Thesis work
+     COSFLEX //Inserted for Thesis work
     } fwd_strategy_t;

     /**
@@ -57,6 +59,8 @@
         CASE(GTMX_PLUS);
         CASE(GRTR_SORT);
         CASE(GRTR_MAX);
+     CASE(COS);     //Inserted for Thesis Work
+     CASE(COSFLEX); //Inserted for Thesis Work
 #undef CASE
         default: return "Unknown forwarding strategy";
         }
@@ -193,6 +197,42 @@
     const Table* remote_; ///< list of routes known by peer node
 }; // class FwdStrategyCompGRTRMAX

+/* COS forwarding strategy added for thesis work by LCDR Christopher
Rapin January
+ * 2011.
+*/
+
+class FwdStrategyCompCOS : public FwdStrategyComp
+{
+public:
+    //
+    // Destructor
+    //
+    virtual ~FwdStrategyCompCOS() {}
+
+    virtual bool operator() (const Bundle* a, const Bundle* b) const
+    {
+
+     //NEED TO ADD COMAPARATOR FOR BUNDLE QOS VALUES
+     //FIFO ordering for bundles of equal priority
+        if (a->priority() == b->priority())
+           return *b < *a;
+     else //sort by priority for bundles of differing COS
+           return a->priority() < b->priority();
+    }
+
+
+protected:
+    friend class FwdStrategy; ///< for factory method
+
+    //
+    // Constructor is protected to restrict access to factory method
+    //
+    FwdStrategyCompCOS(FwdStrategy::fwd_strategy_t fs)
+        : FwdStrategyComp(fs) {}
+
+}; // class FwdStrategyCompCOS
+
+
 /**
  * Due to extensive use of copy constructors in the STL, any
inheritance
  * hierarchy of comparators will always get "clipped" back to the base
@@ -242,6 +282,15 @@
             f = new FwdStrategyCompGRTRMAX(fs,remote);
             break;
         }
+/* Added for thesis work by LCDR Christopher Rapin January 2011. */
+
+     case FwdStrategy::COS:
+     case FwdStrategy::COSFLEX:
+     {
+            f = new FwdStrategyCompCOS(fs);
+            break;
+     }
+//END ADDED CODE
         case FwdStrategy::INVALID_FS:
         default:
             break;
```

H.   DECIDER.H

```
--- orig_Decider.h      2010-02-21 08:11:18.000000000 -0800
+++ Decider.h     2011-02-26 12:54:58.000000000 -0800
@@ -221,6 +221,51 @@
     u_int max_fwd_; ///< local configuration setting for NF_max
 }; // class FwdDeciderGTMXPLUS

+// class FwdDeciderCOSFLEX
+
+/**
+ * Added for Thesis work by LCDR Christopher Rapin January 2010
+ * Forward the bundle only if
+ * Priority is high && P(B,D) > P(A,D)
+ *   -OR-
+ * Priority is Medium && P(B,D) > P(A,D) && NF < Ceil(log2(#nodes))
+ *   -OR-
+ * Priority is Low && P(B,D) > P(A,D) && NF < Ceil(log10(#nodes))
+ * which is a combination of COS(GRTR) and GTMX
+ */
+class FwdDeciderCOSFLEX : public FwdDeciderGRTR
+{
+public:
+    /**
+     * Destructor
+     */
+    virtual ~FwdDeciderCOSFLEX() {}
+
+    /**
+     * Virtual from Decider
+     */
+    bool operator() (const Bundle*) const;
+
+    ///@{ Accessors
+    u_int max_forward() const { return max_fwd_; }
+    ///@}
+protected:
+    friend class Decider; // for factory method
+
+    /**
+     * Constructor.  Protected to force entry via factory method.
+     */
+    FwdDeciderCOSFLEX(FwdStrategy::fwd_strategy_t fs,
+                        const Link* nexthop, BundleCore* core,
+                        const Table* local, const Table* remote,
+                        const Stats* stats,
+                        u_int max_forward, bool relay);
+
+    u_int max_fwd_; ///< local configuration setting for NF_max
+}; // class FwdDeciderCOSFLEX
+
+           //END ADDITIONAL CODE
+
 Decider*
 Decider::decider(FwdStrategy::fwd_strategy_t fs,
                  const Link* nexthop,
@@ -244,6 +289,9 @@
         case FwdStrategy::GRTR:
         case FwdStrategy::GRTR_SORT:
         case FwdStrategy::GRTR_MAX:
+                 //Added for thesis
+     case FwdStrategy::COS:  //Class of service sorted list still uses
GRTR
+                 //End Additional code
         {
             d = new FwdDeciderGRTR(fs,nexthop,core,local_nodes,
                                    remote_nodes,NULL,is_relay);
@@ -272,6 +320,15 @@
                                        is_relay);
             break;
         }
+/*Additional Code added for thesis work by LCDR Christopher Rapin
January 2011*/
+     case FwdStrategy::COSFLEX:
+     {
+            if (stats == NULL || max_forward == 0) return NULL;
+            d = new FwdDeciderCOSFLEX(fs,nexthop,core,local_nodes,
+                                       remote_nodes,stats,max_forward,
+                                       is_relay);
+     }
+                 //END ADDITIONAL CODE
         case FwdStrategy::INVALID_FS:
         default:
            break;
```

I. DECIDER.CC

```
--- orig_Decider.cc     2010-02-21 08:11:18.000000000 -0800
+++ Decider.cc    2011-03-03 13:55:54.000000000 -0800
@@ -15,6 +15,7 @@
  */

 #include "Decider.h"
+#include <math.h> //needed for COSFLEX calculations.

 #define LOG(_args...) if (core_ != NULL) core_->print_log("decider", \
         BundleCore::LOG_DEBUG, _args)
@@ -197,4 +198,102 @@
     return num_fwd_ok && p_max_ok;
 }

+/*
+ * Added for Thesis work by LCDR Christopher Rapin January 2011
+*/
+
+FwdDeciderCOSFLEX::FwdDeciderCOSFLEX(FwdStrategy::fwd_strategy_t fs,
+                                       const Link* nexthop,
+                                       BundleCore* core,
+                                       const Table* local,
+                                       const Table* remote,
+                                       const Stats* stats,
+                                       u_int max_forward,
+                                       bool relay)
+    : FwdDeciderGRTR(fs,nexthop,core,local,remote,stats,relay),
+      max_fwd_(max_forward) {}
+
+bool
+FwdDeciderCOSFLEX::operator()(const Bundle* b) const
+{
+    // defer first refusal to base class
+    if (! FwdDeciderGRTR::operator()(b))
+        return false;
+
+    if (core_->is_route(b->destination_id(),next_hop_->nexthop()))
+    {
+        LOG("ok to fwd: %s is destination for bundle %s %u:%u",
+            next_hop_->remote_eid(),b->destination_id().c_str(),
+            b->creation_ts(),b->sequence_num());
+        return true;
+    }
+
+    // if no route match and remote is not a relay,
+    // don't bother forwarding this bundle
+    else if (!is_relay_)
+        return false;
+
+/* Decider uses a boolean variable to determine forwarding.
+*  num_fwd_ok -
+*  checks the number of times the bundle has been forwarded.  For high
priority
```

```
+*  bundles this is always true as they use the GRTR forwarding
strategy with
+*  no restricitons on how many times a bundle can be forwarded.
+*  Medium priority bundles compare it to the ceiling of the natural
logarithm
+*  of NF.  When using this strategy NF should be set to the network
size rather
+*  than the number of times the bundle should be forwarded. For low
priority
+*  bundles (the else case) it is the ceiling of the base ten logarithm
of NF
+*  that is used.  The effect of these modified forwarding strategies
is to
+*  to effectively implement GRTR+ with differing forwarding numbers
for med and
+*  low priority bundles without having to add additional information
to the node
+*  or the bundle being forwarded.
+
+*/
+    std::string dest_eid = core_->get_route(b->destination_id());
+    bool num_fwd_ok;
+    int normal = ceil(log(max_fwd_));//max # of forwards for normal
priority
+    int bulk = ceil(log10(max_fwd_));//max # of forwards for bulk
priority
+
+    if (b->priority() == 2){ //decider and logging for high priority
bundles
+     num_fwd_ok = true; //always true, no forwarding restrictions
+
+         if (num_fwd_ok)
+           LOG("ok to fwd expidited: remote p (%.2f) > "
+                 "local (%.2f) for %s %u:%u",remote_-
>p_value(dest_eid),
+           local_->p_value(dest_eid),
+                 b->destination_id().c_str(), b->creation_ts(),
+                 b->sequence_num());
+
+     return num_fwd_ok;
+    }
+    if (b->priority() == 1){//decider and logging for medium priority
bundles
+     num_fwd_ok = (int)b->num_forward() < normal;
+
+         if (num_fwd_ok)
+             LOG("ok to fwd normal: NF (%u) < Ceil(log(MaxFwd)) (%u) "
+           "and remote p (%.2f) > local (%.2f) for %s %u:%u",
+           b->num_forward(), (int)ceil(log(max_fwd_)),
+                   remote_->p_value(dest_eid), local_-
>p_value(dest_eid),
+                b->destination_id().c_str(), b->creation_ts(),
+                b->sequence_num());
+     return num_fwd_ok;
+    }
+    if (b->priority() == 0){ //decider and logging for low priority
bundles
+     num_fwd_ok = (int)b->num_forward() < bulk;
+
+     if (num_fwd_ok)
+             LOG("ok to fwd bulk: NF (%u) < Ceil(log10(MaxFwd)) (%u) "
+           "and remote p (%.2f) > local (%.2f) for %s %u:%u",
+           b->num_forward(), (int)ceil(log10(max_fwd_)),
+                   remote_->p_value(dest_eid), local_-
>p_value(dest_eid),
+                b->destination_id().c_str(), b->creation_ts(),
+                b->sequence_num());
+
+     return num_fwd_ok;
+    }
+
+    return false;
+
+}
+           /* End additional Thesis code */
+
 }; // namespace prophet
```

J. QUEUEPOLICY.H

```
--- orig_QueuePolicy.h  2010-02-21 08:11:18.000000000 -0800
+++ QueuePolicy.h 2011-02-03 09:01:48.000000000 -0800
@@ -42,7 +42,8 @@
         MOPR,
         LINEAR_MOPR,
         SHLI,
-        LEPR
+        LEPR,
+     PRIORITY //Added for Thesis Work
     } q_policy_t;

     /**
@@ -59,6 +60,7 @@
         CASE(LINEAR_MOPR);
         CASE(SHLI);
         CASE(LEPR);
+     CASE(PRIORITY); //Added for Thesis Work
 #undef CASE
         default: return "Unknown queuing policy";
         }
@@ -80,6 +82,8 @@
             return SHLI;
         if (qp == "LEPR")
             return LEPR;
+     if (qp == "PRIORITY") //Added for Thesis Work
+         return PRIORITY;  //Added for Thesis Work
         return INVALID_QP;
     }

@@ -347,6 +351,50 @@
```

```
         : QueueComp(qp,NULL,nodes,min_forward) {}
 }; // class QueueCompLEPR

+/**
+ * Queuing policy comparator for PRIORITY added for Thesis work by
LCDR Christopher
+ * Rapin January 2011
+ */
+class QueueCompPRIORITY : public QueueComp
+{
+public:
+    /**
+     * Destructor
+     */
+    virtual ~QueueCompPRIORITY() {}
+
+    /**
+     * Virtual from std::greater
+     */
+    virtual bool operator() (const Bundle* a, const Bundle* b) const
+    {
+        // evict oldest bundle with the lowest priority
+        if (verbose_)
+        printf("PRIORITY: %d (%d) %s %d (%d)\n",
+               a->sequence_num(),
+               a->priority(),
+               (a->priority() < b->priority()) ? ">" : "<",
+               b->sequence_num(),
+               b->priority());
+
+        //FIFO ordering for bundles of equal priority
+        if (a->priority() == b->priority())
+           return *b < *a;
+     else
+           return a->priority() < b->priority();
+
+    }
+
+protected:
+    friend class QueuePolicy;
+
+    /**
+     * Constructor, protected to enforce factory method
+     */
+    QueueCompPRIORITY(QueuePolicy::q_policy_t qp)
+        : QueueComp(qp) {}
+}; // class QueueCompPRIORITY
+
 QueueComp*
 QueuePolicy::policy(QueuePolicy::q_policy_t qp,
                     const Stats* stats, const Table* nodes,
@@ -390,6 +438,12 @@
            return new QueueCompLEPR(qp,nodes,min_forward);
         }

+/* Added for thesis work by LCDR Christopher Rapin January 2011. */
```

```
+     //evict oldest bundle with the lowest priority first.
+     case QueuePolicy::PRIORITY:
+         return new QueueCompPRIORITY(qp);
+//END ADDED CODE
+
         // oops, no QP specified
         case QueuePolicy::INVALID_QP:
         default:
@@ -397,6 +451,8 @@
     }
 }

+
+
 }; // namespace prophet

 #endif // _PROPHET_QUEUE_POLICY_H_
```

K. BUNDLE.H

```
--- orig_Bundle.h 2010-02-21 08:11:18.000000000 -0800
+++ Bundle.h      2011-01-27 10:59:46.000000000 -0800
@@ -45,6 +45,10 @@
     virtual       u_int        size()              const = 0;
     virtual       u_int        num_forward()       const = 0;
     virtual       bool         custody_requested() const = 0;
+//Additional Code added for Thesis work by LCDR Christopher Rapin
January 2011
+    virtual        u_int8_t     priority()        const = 0;
+//END ADDITONAL CODE
+
     ///@}

     ///@{ Operators
```

L. BUNDLEIMPL.H

```
--- orig_BundleImpl.h   2010-02-21 08:11:18.000000000 -0800
+++ BundleImpl.h  2011-01-27 10:13:28.000000000 -0800
@@ -43,7 +43,10 @@
           ets_(0),
           size_(0),
           num_fwd_(0),
-          custody_requested_(false)
+          custody_requested_(false),
+//Code added for Thesis work by LCDR Christopher Rapin January 2011
+       priority_(0)
+//end added code
     {}

     /**
@@ -55,7 +58,10 @@
            u_int32_t expiration_ts = 0,
            u_int size = 0,
            u_int num_forward = 0,
-           bool custody_requested = false)
+           bool custody_requested = false,
+//Code added for thesis work by LCDR Christopher Rapin January 2011
+            u_int8_t priority = 0)
+//end added code
         : Bundle(),
           dest_id_(destination_id),
           src_id_(""),
@@ -64,7 +70,10 @@
           ets_(expiration_ts),
           size_(size),
           num_fwd_(num_forward),
-          custody_requested_(custody_requested)
+          custody_requested_(custody_requested),
+//Code added for thesis work by LCDR Christopher Rapin January 2011
+       priority_(priority)
+//end added code
     {}

     /**
@@ -77,7 +86,10 @@
            u_int32_t expiration_ts = 0,
            u_int size = 0,
            u_int num_forward = 0,
-           bool custody_requested = false)
+           bool custody_requested = false,
+//Code added for thesis work by LCDR Christopher Rapin January 2011
+        u_int8_t priority = 0)
+//end added code)
         : Bundle(),
           dest_id_(destination_id),
           src_id_(source_id),
@@ -86,7 +98,10 @@
           ets_(expiration_ts),
           size_(size),
           num_fwd_(num_forward),
-          custody_requested_(custody_requested)
+          custody_requested_(custody_requested),
+//Code added for thesis work by LCDR Christopher Rapin January 2011
+       priority_(priority)
+//end added code
     {}

     /**
@@ -98,7 +113,10 @@
           src_id_(b.src_id_), cts_(b.cts_),
           seq_(b.seq_), ets_(b.ets_),
           size_(b.size_), num_fwd_(b.num_fwd_),
-          custody_requested_(b.custody_requested_)
+          custody_requested_(b.custody_requested_),
+//Code added for thesis work by LCDR Christopher Rapin January 2011
+       priority_(b.priority_)
+//end added code
     {}

     /**
```

```
@@ -115,6 +133,9 @@
     virtual u_int       size()           const { return size_; }
     virtual u_int       num_forward()    const { return num_fwd_; }
     virtual bool     custody_requested() const { return
custody_requested_; }
+//Additional Code added for Thesis work by LCDR Christopher Rapin
January 2011
+    virtual u_int8_t     priority()         const {return priority_;}
+//END ADDITONAL CODE
     ///@}

     ///@{ Mutators
@@ -126,6 +147,9 @@
     virtual void set_size( u_int sz ) { size_ = sz; }
     virtual void set_num_forward( u_int nf ) { num_fwd_ = nf; }
     virtual void set_custody_requested( bool c ) { custody_requested_
= c; }
+//Additional Code added for Thesis work by LCDR Christopher Rapin
January 2011
+    virtual void set_priority( u_int8_t priority ) { priority_ =
priority; }
+//END ADDITONAL CODE
     ///@}

     ///@{ Operators
@@ -138,6 +162,9 @@
         ets_     = b.ets_;
         size_    = b.size_;
         num_fwd_ = b.num_fwd_;
+//Additional Code added for Thesis work by LCDR Christopher Rapin
January 2011
+     priority_ = b.priority_;
+//END ADDITONAL CODE
         return *this;
     }
     ///@}
@@ -151,6 +178,9 @@
     u_int     size_;      ///< size of Bundle payload
     u_int     num_fwd_;   ///< number of times this Bundle has been
forwarded
     bool custody_requested_; ///< whether to request custody on this
bundle
+//Additional Code added for Thesis work by LCDR Christopher Rapin
January 2011
+    u_int8_t  priority_;  ///< The CoS priority of this bundle.
+//END ADDITONAL CODE
 }; // class BundleImpl

 }; // namespace prophet
```

M. NODE.CC

```
--- orig_Node.cc  2010-02-21 08:11:18.000000000 -0800
+++ Node.cc 2011-03-03 13:50:38.000000000 -0800
@@ -14,13 +14,17 @@
```

```
  *    limitations under the License.
  */

-#include "Node.h"
+#include "Node.h"      // includes RIBNodeList.
 #include <time.h>       // for time()
 #include <netinet/in.h> // for ntoh*,hton*
 #include <math.h>       // for pow()

+
+
+
 namespace prophet {

+
 const double
 NodeParams::DEFAULT_P_ENCOUNTER = 0.75;

@@ -30,8 +34,11 @@
 const double
 NodeParams::DEFAULT_GAMMA = 0.99;

+//Hardcoded Kappa to prevent unpredictable performance.
+//Code changed by LCDR Christopher Rapin Jan 2011 for thesis work.
 const u_int
-NodeParams::DEFAULT_KAPPA = 100;
+NodeParams::DEFAULT_KAPPA = 3000;
+//End modified code.

 Node::Node(const NodeParams* params)
     : p_value_(0.0), relay_(DEFAULT_RELAY), custody_(DEFAULT_CUSTODY),
@@ -114,6 +121,7 @@
     // C is the node represented by this ProphetNode instance
     p_value_ = (p_value_ * params_->beta_) + (1.0 - params_->beta_) *
ab * bc * params_->encounter_;

+
     // update age to reflect data "freshness"
     age_ = time(0);
 }
@@ -128,11 +136,13 @@
     // new p_value_ is P_(A,B), previous p_value is P_(A,B)_old
     // params_->gamma_ is gamma
     // timeunits is k
+
     u_int32_t now = time(0);

     double agefactor = 1.0;

     u_int timeunits = time_to_units(now - age_);
+
     if (timeunits > 0)
         agefactor = pow( params_->gamma_, timeunits );
```

N. CONTROLLER.CC

```
--- orig_Controller.cc  2010-02-21 08:11:18.000000000 -0800
+++ Controller.cc 2011-02-17 18:54:44.000000000 -0800
@@ -177,9 +177,13 @@
 {
     if (b == NULL) return;

-    LOG(LOG_DEBUG,"handle_bundle_received(%d,%s)",
-            b->sequence_num(), (link == NULL) ? "NULL" : link-
>remote_eid());
-
+/*Code removed to prevent a segmentation fault when the bundle being
added
+**is immediately dropped because it is the lowest priority bundle in a
queue
+**that is over quota.
+**
+**  LOG(LOG_DEBUG,"handle_bundle_received(%d,%s)",
+**          b->sequence_num(), (link == NULL) ? "NULL" : link-
>remote_eid());
+*/
     if (link == NULL)
         return;
```

O. REPOSITORY.CC

```
--- orig_Repository.cc  2010-02-21 08:11:18.000000000 -0800
+++ Repository.cc 2011-03-04 23:46:00.000000000 -0800
@@ -15,6 +15,7 @@
  */

 #include "Repository.h"
+#include "storage/BundleStore.h"

 #define LOG(_level,_args...) core_->print_log("repository", \
         BundleCore::_level, _args);
@@ -95,12 +96,15 @@
     // reorder sequence to eviction order
     size_t last_pos = list_.size() - 1;
     push_heap(0,last_pos,0,list_[last_pos]);
-    // increment utilization by this Bundle's size
-    current_ += b->size();
+    // update utilization from DTN's BundleStore
+    dtn::BundleStore* bs = dtn::BundleStore::instance();
+    current_ = bs->total_size();
     // maintain quota
+    LOG(LOG_INFO, "Bundle successfully added to prophet repository")
     if (core_->max_bundle_quota() > 0)
         while (core_->max_bundle_quota() < current_)
             evict();
+    LOG(LOG_INFO, "RAPIN Queue size: %d", list_.size())
     return true;
 }
```

```
@@ -164,13 +168,10 @@
         pop_heap(0, last_pos, last_pos, list_[last_pos]);
         // capture a pointer to the back of list_
         const Bundle* b = list_.back();
-        // drop the last member off the end of list_
-        list_.pop_back();
-        // callback into Bundle core to request deletion of Bundle
-        core_->drop_bundle(b);
-        // decrement current consumption by Bundle's size
-        current_ -= b->size();
-
+
+        // drop the last member off the end of list_
+     core_->drop_bundle(b);
+LOG(LOG_INFO, " RAPIN *** Prophet Evicted the previous drop due to
being over storage quota!")
         return;
     }
     else
@@ -185,13 +186,10 @@
             if (comp_->min_fwd_ < b->num_forward())
             {
                 // victim is found, now evict
-                // decrement utilization by Bundle's size
-                current_ -= b->size();
+                core_->drop_bundle(b);
                 // reorder sequence to preserve eviction ordering,
moving
                 // victim to last pos in vector
                 remove_and_reheap(pos);
-                // remove victim from vector
-                list_.pop_back();
                 return;
             }
         }
@@ -203,6 +201,7 @@
         // top of that.
         goto do_evict;
     }
+
 }

 void
```

P. PROPHETBUNDLE.H

```
--- orig_ProphetBundle.h      2010-02-21 08:11:18.000000000 -0800
+++ ProphetBundle.h     2011-01-27 11:02:32.000000000 -0800
@@ -102,6 +102,12 @@
     {
         return (ref_ == NULL) ? false : ref()->custody_requested();
     }
+/*Added for Thesis work by LCDR Christopher Rapin January 2011 */
+    virtual u_int8_t priority() const
+    {
```

```
+     return (ref_==NULL) ? 0 : ref()->priority();
+    }
+//END additional code
     ///@}

 protected:
```

Q. PROPHETBUNDLECORE.CC

```
--- orig_ProphetCommand.cc    2010-02-21 08:11:18.000000000 -0800
+++ ProphetCommand.cc   2011-02-02 12:13:12.000000000 -0800
@@ -99,7 +99,10 @@
                 "\tmopr\tevict most favorably forwarded first\n"
                 "\tlmopr\tevict most favorably forwarded first (linear
increase)\n"
                 "\tshli\tevict shortest lifetime first\n"
-                "\tlepr\tevice least probable first\n");
+                "\tlepr\tevice least probable first\n"
+/*Added for thesis work by LCDR Christopher Rapin January 2011*/
+                "\tpriority\tevict oldest lowest priority first\n");
+//END ADDED CODE

     add_to_help("fwd_strategy=<strategy>",
                 "set forwarding strategy to one of the following:\n"
@@ -108,7 +111,11 @@
                 "\tgrtr_plus\tforward if \"grtr\" and P > P_Max\n"
                 "\tgtmx_plus\tforward if \"grtr_plus\" and NF <
NF_Max\n"
                 "\tgrtr_sort\tforward if \"grtr\" and sort desc by
P_remote - P_local\n"
-                "\tgrtr_max\tforward if \"grtr\" and sort desc by
P_remote\n");
+                "\tgrtr_max\tforward if \"grtr\" and sort desc by
P_remote\n"
+/*Added for thesis work by LCDR Christopher Rapin January 2011*/
+                "\tcos\tforward if remote's P is greater and sort by
class of service\n"
+                "\tcosflex\tforward if remote's P is greater and sort
by class of service\n         with restrictions on forwarding.
Expedited CoS - no restrictions on the \n             number of times a
bundle is forwarded.  Normal CoS - Restricted to being\n
      forwarded no more than ln(x) times, where x is the network size
provided \n       by the max_forward field.  Bulk CoS - Restricted to
being forwarded no   \n             more than Log10(x) where x is the
network size provided in the           \n             max_forward
field.\n");
+//END ADDED CODE

     add_to_help("hello_interval=<interval>",
                 "maximum delay between protocol messages, in 100ms
units,"
@@ -150,6 +157,10 @@
             {"gtmx_plus", prophet::FwdStrategy::GTMX_PLUS},
             {"grtr_sort", prophet::FwdStrategy::GRTR_SORT},
             {"grtr_max",  prophet::FwdStrategy::GRTR_MAX},
+/*Added for Thesis work by LCDR Christopher Rapin January 2011 */
+         {"cos",   prophet::FwdStrategy::COS},
+         {"cosflex",     prophet::FwdStrategy::COSFLEX},
+//END ADDED CODE
             {0, 0}
         };
         int fs_pass = ProphetRouter::params_.fs_;
@@ -179,6 +190,9 @@
             {"lmopr", prophet::QueuePolicy::LINEAR_MOPR},
             {"shli",  prophet::QueuePolicy::SHLI},
             {"lepr",  prophet::QueuePolicy::LEPR},
+/*Added for Thesis work by LCDR Christopher Rapin January 2011*/
+            {"priority",  prophet::QueuePolicy::PRIORITY},
+//END ADDED CODE
             {0, 0}
         };
         int qp_pass;
```

R. PROPHETROUTER.CC

```
--- orig_ProphetRouter.cc     2010-02-21 08:11:18.000000000 -0800
+++ ProphetRouter.cc    2011-02-25 13:47:04.000000000 -0800
@@ -145,12 +145,18 @@
 {
     log_info("ProphetRouter accept_bundle");

+//Code removed.  We dont want prophet asking the base class, we want
prophet to call
+//its queueing strategy to decide.  So we accept the bundle in all
cases and then, if we
+//are over quota we call evict() to handle it.
+
     // first ask base class
-    if (!BundleRouter::accept_bundle(bundle,errp))
+/*    if (!BundleRouter::accept_bundle(bundle,errp))
     {
         log_debug("BundleRouter rejects *%p",bundle);
         return false;
-    }
+    }*/
+
+//END MODIFIED CODE

     BundleRef tmp("accept_bundle");
     tmp = bundle;
@@ -184,6 +190,7 @@
     // [Note from Elwyn Davies: Maybe using a special link might be
useful]
     if ((e->source_ != EVENTSRC_APP) && (e->source_ !=
EVENTSRC_ADMIN))
     {
+
      // The external CL does not set this field, which the Prophet
      // implementation needs. We want to fail quickly if we're
      // running with the ECL.
@@ -208,6 +215,11 @@
         return;
     }

+//Code added for thesis work by LCDR Christopher Rapin February 2011//
+    if (e->source_ == EVENTSRC_APP)
+     log_info("RAPIN BUNDLE CREATION!! ID: %u Priority: %u Creation
time: %u",
+           b->sequence_num(), b->priority(), b->creation_ts());
+//end additional code
     core_->bundles_.add(b);

     // inform Controller that a new bundle has arrived on this link
```

S. PROPHETCOMMAND.CC

```
--- orig_ProphetCommand.cc 2010-02-21 09:11:18.000000000 -0800
+++ ProphetCommand.cc 2011-02-02 13:13:12.000000000 -0800
@@ -99,7 +99,10 @@
                 "\tmopr\tevict most favorably forwarded first\n"
                 "\tlmopr\tevict most favorably forwarded first
(linear increase)\n"
                 "\tshli\tevict shortest lifetime first\n"
-                "\tlepr\tevice least probable first\n");
+                "\tlepr\tevice least probable first\n"
+/*Added for thesis work by LCDR Christopher Rapin January 2011*/
+                "\tpriority\tevict oldest lowest priority
first\n");
+//END ADDED CODE

     add_to_help("fwd_strategy=<strategy>",
                 "set forwarding strategy to one of the
following:\n"
@@ -108,7 +111,11 @@
                 "\tgrtr_plus\tforward if \"grtr\" and P >
P_Max\n"
                 "\tgtmx_plus\tforward if \"grtr_plus\" and NF <
NF_Max\n"
                 "\tgrtr_sort\tforward if \"grtr\" and sort desc
by P_remote - P_local\n"
-                "\tgrtr_max\tforward if \"grtr\" and sort desc
by P_remote\n");
+                "\tgrtr_max\tforward if \"grtr\" and sort desc
by P_remote\n"
+/*Added for thesis work by LCDR Christopher Rapin January 2011*/
+                "\tcos\tforward if remote's P is greater and
sort by class of service\n"
+                "\tcosflex\tforward if remote's P is greater and
sort by class of service\n      with restrictions on forwarding.
Expedited CoS - no restrictions on the \n       number of times
a bundle is forwarded.  Normal CoS - Restricted to being\n
     forwarded no more than ln(x) times, where x is the network
```

```
size provided \n      by the max_forward field.  Bulk CoS -
Restricted to being forwarded no   \n      more than Log10(x)
where x is the network size provided in the          \n
     max_forward field.\n");
+//END ADDED CODE

     add_to_help("hello_interval=<interval>",
                "maximum delay between protocol messages, in
100ms units,"
@@ -150,6 +157,10 @@
             {"gtmx_plus", prophet::FwdStrategy::GTMX_PLUS},
             {"grtr_sort", prophet::FwdStrategy::GRTR_SORT},
             {"grtr_max",  prophet::FwdStrategy::GRTR_MAX},
+/*Added for Thesis work by LCDR Christopher Rapin January 2011
*/
+        {"cos",       prophet::FwdStrategy::COS},
+        {"cosflex",   prophet::FwdStrategy::COSFLEX},
+//END ADDED CODE
             {0, 0}
         };
         int fs_pass = ProphetRouter::params_.fs_;
@@ -179,6 +190,9 @@
             {"lmopr", prophet::QueuePolicy::LINEAR_MOPR},
             {"shli",  prophet::QueuePolicy::SHLI},
             {"lepr",  prophet::QueuePolicy::LEPR},
+/*Added for Thesis work by LCDR Christopher Rapin January 2011*/
+            {"priority",  prophet::QueuePolicy::PRIORITY},
+//END ADDED CODE
             {0, 0}
         };
         int qp_pass;
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] T. Spyropoulos, K. Psounis, and C. Raghavendra. “Efficient routing in intermittently connected mobile networks: the multiple-copy case.” *IEEE/ACM Trans. Netw.* 16, 1 (Feb. 2008), 77-90.

[2] T. Spyropoulos, K. Psounis, and C. Raghavendra. “Efficient routing in intermittently connected mobile networks: the single-copy case.” *IEEE/ACM Trans. Netw. 16,* 1 (Feb. 2008), 63-76.

[3] A. Lindgren, A. Doria, E. Davies, and S. Grasic. “Probabilistic Routing Protocol for Intermittently Connected Networks,” work in progress draft-IRTF-DTNRG-prophet-07 Aug. 2010 expires Feb. 2011.

[4] J. Burgess, B. Gallagher, D. Jensen, and B. Levine. "MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks," *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pp. 1-11, April 2006

[5] T. Spyropoulos, K. Psounis, and C. Raghavendra. “Spray and wait: an efficient routing scheme for intermittently connected mobile networks.” *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking* (WDTN '05). ACM, New York, NY, USA, 252-259.

[6] The ByteWalla Group. Available at: http://www.tslab.ssvl.kth.se/csd/projects/092106/home.

[7] S. Farrell. “Delay Tolerant Networking Research Group,” Feb. 11 2011. Available at: http://www.dtnrg.org/wiki.

[8] F. Warthman. “Delay Tolerant Networks (DTNs) A Tutorial.” Available at: *http://*www.dtnrg.org/docs/tutorials/warthman-1.1.pdf. Mar. 5 2003.

[9] K. Scott, S. Burleigh. “Network Working Group RFC 5050 Bundle Protocol Specification,” November 2007. Available at: www.rfc-editor.org/rfc/rfc5050.txt*.*

[10] A. Vahdat, and D. Becker. Epidemic routing for partially-connected ad hoc networks. Technical report, Duke University, 2000.

[11] R. Yanggratoke, A. Azfar, M. Maraval, and S. Ahmed. “ByteWalla: Delay Tolerant Networks on Android Phones,” Fall 2009. Available at: http://www.tslab.ssvl.kth.se/csd/projects/092106/sites/default/files/Bytewalla%20System%20Architecture%20Design%20v1.0%202009.09.15.pdf.

[12] DTN and Oasys source code. “Delay Tolerant Networking – Browse Files at SourceForge.net.” Available at: http://sourceforge.net/projects/dtn/files/

[13] DTN User Manual. “Table of Contents” Available at: http://dtn.sourceforge.net/DTN2/doc/manual/index.html

[14] NielsenWire. “In U.S., SMS Text Messaging Tops Mobile Phone Calling,” September 22, 2008. Available at: http://blog.nielsen.com/nielsenwire/online_mobile/in-us-text-messaging-tops-mobile-phone-calling/

[15] Ubuntu 10.04 Source Code. “Ubuntu 10.04 LTS – Long Term Support,” 2010. Available at: http://www.ubuntu.com/desktop/get-ubuntu/download

[16] Berkeley DB Source Code. “Berkeley DB 4.7.25NC.tar.gz” Available at: http://www.oracle.com/technetwork/database/berkeleydb/downloads/index-082944.html

[17] Oasys Source Code. “oasys-1.4.0.” Available at: http://sourceforge.net/projects/dtn/files/oasys/oasys-1.4.0/

[18] DTN2 Source Code “dtn-2.7.0.” Available at: http://sourceforge.net/projects/dtn/files/DTN2/dtn-2.7.0/

[19] K. Fall, A. McMahon. “DTN Reference Implementation Update,” November 13, 2009. Available at: http://www.ietf.org/proceedings/76/slides/DTNRG-5.pdf.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Geoffrey Xie
   Naval Postgraduate School
   Monterey, California

4. Robert Beverly
   Naval Postgraduate School
   Monterey, California

5. Jerome Rapin
   OPNAV N2/N6F3B
   Navy Pentagon, 1E236
   Washington, DC